PARSEVAL: parallelisation sur reseaux de transputers de simulations pour l'evaluation de performances

Hery Rakotoarisoa, Philippe Mussi

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports Techniques

N°131

Programme 1
Architectures parallèles, Bases de données, Réseaux et Systèmes distribués

PARSEVAL: PARallélisation sur réseaux de *Transputers* de Simulations pour l'EVALuation de performances

Hery RAKOTOARISOA
Philippe MUSSI

Septembre 1991

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél : 93.65.77.77

PARSEVAL:
PARallélisation sur réseaux de *Transputers* de Simulations pour l'ÉVALuation de performances

Parallelising Performance Evaluation Simulations on Transputers Networks

Hery RAKOTOARISOA
Philippe MUSSI

INRIA – Centre de Sophia Antipolis
2004, Route des Lucioles– BP 109
06561 Valbonne Cedex – France

Abstract

This report describes PARSEVAL, an experimental distributed simulator of queueing systems running on a network of *Transputers*. It is an implementation of the *conservative method* proposed by Chandy and Misra for the synchronisation of processes.

The sequential and the distributed discrete-event simulations are first presented in order to show the main problem which is the synchronisation. After the description of the conservative method, the structure of PARSEVAL and the way it works are detailed, and this report ends with some consideration on its performance compared to the sequential simulator QNAP2.

Résumé

Ce rapport décrit PARSEVAL, un simulateur distribué expérimental de réseaux de files d'attente, fonctionnant sur un réseau de *Transputers.* C'est une implémentation de la méthode de synchronisation de processus, dite "conservative", proposée par Chandy et Misra.

Les simulations à événements discrets séquentielle et distribuée sont d'abord présentées pour bien montrer le problème principal qu'est la synchronisation. Après la description de la méthode conservative, la structure de PARSEVAL et son fonctionnement sont détaillés, et ce rapport se termine par une étude de ses performances comparées à celles du simulateur séquentiel QNAP2.

Introduction

On cherche à distribuer la simulation de modèles de réseaux de files d'attente sur une architecture multiprocesseur, en l'occurence un réseau de transputers[1], le but étant d'utiliser de façon optimale la configuration matérielle disponible pour améliorer les performances. La simulation à événements discrets classique est donc présentée en premier lieu pour bien cadrer l'objet de l'étude et pouvoir passer par la suite à la simulation distribuée.

La simulation d'une partie d'un réseau de files d'attente est affectée à un processeur qui ne communique avec les autres que par échanges de messages, sur les liens physiques dans le cas des transputers, sur un réseau d'interconnexion quelconque dans le cas général. Chaque processeur est donc indépendant et travaille de manière totalement asynchrone par rapport aux autres, posant ainsi le problème principal: leur "*synchronisation*" durant la simulation.

Différentes méthodes ont été proposées dans la littérature pour réaliser cette synchronisation, notamment la méthode dite "*time warp*" [6], et celle dite "*conservative*" [4]. La première est classée comme une méthode optimiste car elle consiste à faire progresser la simulation au sein de chaque processus sans se préoccuper de la vitesse des autres. C'est seulement aux moments où un processus reçoit un message lui indiquant qu'il est parti trop loin dans la simulation qu'il va revenir en arrière et annuler les traitements erronés déjà effectués. Il peut donc y avoir une différence importante entre les dates courantes de simulation des processus. La deuxième consiste à bloquer un processus jusqu'à ce que les autres lui indiquent qu'il peut avancer. L'ensemble progresse ainsi à la même vitesse et la différence de date entre les processus à un instant donné est minime.

C'est cette méthode qui est implémentée dans le simulateur expérimental nommé PARSEVAL[2], objet de ce rapport. Après la présentation de la méthode conservative, la structure et le fonctionnement de ce simulateur sont détaillés. Les résultats sont ensuite comparés à ceux du simulateur à événements discrets de QNAP2 [11] pour leur validation et pour la considération des performances de PARSEVAL, et donc de la méthode conservative. Ce rapport se termine par la discussion des améliorations nécessaires pour augmenter ces performances.

---

[1] Le transputer, fabriqué par INMOS, est un processeur conçu pour la programmation parallèle et équipé de quatre liens bidirectionnels lui permettant de s'insérer dans un réseau (voir [15]).

[2] PARallélisation de Simulations pour l'ÉVALuation de performances.

Chapitre 1

La simulation à événements discrets

1.1 La simulation à événements discrets classique

(d'après [1])

On appelle "*événement*" le changement d'état du système, provoqué par l'apparition de données venant de l'extérieur, ou résultant de ses traitements internes. La simulation à événements discrets a les caractéristiques suivantes :

- Les "*variables d'état*" sont discrètes.
- Les événements sont discrets, c'est à dire qu'ils apparaissent à des instants discrets (l'état du système ne peut changer de manière continue). Ces instants d'occurrence des événements sont appelés "*dates*" des événements.
- Les variables continues (y compris le temps) ne sont prises en compte qu'aux instants où elles sont utilisées, c'est à dire aux moments d'apparition des événements. On dit alors qu'elles sont discrétisées et on les appelle "*variables déduites*" ou "*variables annexes*"

Il existe deux types de simulation à événements discrets: celle dirigée par horloge et celle dirigée par événements.

1.1.1 La simulation dirigée par horloge

Cette simulation entretient une horloge centrale, qui progresse par unité de temps, de longueur fixée selon le problème traité. A chaque incrémentation de l'horloge, la liste des événements est parcourue, et chaque événement portant la date correspondante à l'horloge est simulé. Il est évident qu'il peut y avoir échec dans ce parcours de la liste, si aucun événement ne correspond à la date courante. Le choix judicieux de l'unité de temps en fonction du problème à simuler est donc fondamental dans ce type de simulation. Nous n'allons pas développer en détail cette simulation[1] car ce n'est pas l'objet de cette étude.

1.1.2 La simulation dirigée par événements

Dans ce type de simulation, il n'existe pas d'horloge centrale. Les paramètres qui représentent le temps sont les dates d'apparition des événements. La progression temporelle de la simulation se fait donc au fur et à mesure que le système avance dans le traitement des événements. Pour gérer les événements, et donc le temps, on utilise une liste appelée "*échéancier*". Les événements y sont rangés par ordre croissant de date. Ainsi, en tête de la liste se trouve celui en cours de traitement, portant la date la plus petite, et à la fin de la liste celui portant la date la plus grande.

[1] bien qu'elle présente un intérêt certain sur machines parallèles

Remarque :
Tous les événements autres que celui en tête de la liste, sont en fait potentiels car ils peuvent être décalés, modifiés ou supprimés durant la simulation de celui en cours.

On appelle "*noyau de synchronisation*" l'ensemble des procédures d'entretien de l'échéancier (ajout, suppression et modification d'éléments). L'occurrence d'un événement, suivi de son traitement, modifiant l'état du système (donc l'échéancier et la date de la simulation), constitue une "*activité*" de la simulation. Il existe deux techniques de réalisation du noyau de synchronisation : celle basée sur la notion d'événements et celle basée sur la notion de processus.

1.1.2.1 La simulation basée sur la notion d'événements

C'est le type de l'événement qui définit l'action de la simulation. L'arrivée d'un événement donné en tête de l'échéancier, provoque l'exécution de la procédure correspondante fournie par le noyau de synchronisation. Ceci peut éventuellement ordonnancer d'autres événements. Cette technique est adaptée à la simulation d'un problème posé en terme d'événements, par exemple un programme qui se branche sur une routine après l'apparition d'une interruption.

1.1.2.2 La simulation basée sur la notion de processus

Ici, chaque activité de la simulation est considérée comme l'activité d'un processus. Ainsi, la simulation est constituée d'un ensemble de processus s'exécutant en parallèle. Le noyau de synchronisation n'est plus un ensemble de procédures, comme dans le cas précédent, mais sera un ensemble de primitives de manipulation de processus telles que *activer*, *suspendre* etc.

Cette technique est adaptée à la simulation d'un problème posé en terme d'activités se déroulant en parallèle. De plus, elle permet la distinction et la dualité entre deux entités dans le système :

- les entités passives qui sont les messages, appelés aussi "*clients*", circulant entre les processus, et
- les entités actives qui sont les processus, appelés "*serveurs*", traitant ces messages.

1.2 La simulation distribuée

La structure même de l'échéancier reflète la nature séquentielle du système, et impose un traitement des événements les uns après les autres. Ainsi, la simulation à événements discrets d'un système a été traditionnellement réalisée de manière séquentielle. Ceci s'avère de plus en plus pénalisant quant à la rapidité, et donc les performances globales de la simulation, car plus le système est complexe, plus la quantité d'événements à traiter augmente. Considérant les possibilités offertes actuellement par les systèmes informatiques, l'idée est de partitionner la simulation sur plusieurs ordinateurs ou plusieurs processeurs reliés en réseau (figure 1.1). Ceci impose de mettre en place de nouvelles structures de données et de nouveaux algorithmes car, un seul échéancier et une seule variable contenant la date de la simulation, comme dans le cas de la simulation séquentielle, ne suffisent plus.

La modélisation d'un système à l'aide de réseaux de files d'attente permet de le décomposer en plusieurs activités indépendantes, communiquant par des messages, qui sont également appelés "*clients*", chacune des activités étant représentée par une file munie de son serveur. Chaque file sera alors simulée par un processus et la simulation sera constituée de plusieurs processus s'exécutant en parallèle.

Les messages circulant dans la simulation sont l'image de ceux existant dans le système réel. Supposons qu'aux instants t_1, t_2, ..., t_p tels que $0 < t_1 < t_2 < ... < t_p$, un processus P_i du système réel envoie respectivement les messages m_1, m_2, ..., m_p au processus P_j (messages ordonnés dans le temps), le simulateur de P_i doit envoyer les mêmes messages à celui simulant P_j (et dans le "bon" ordre). Le problème étant posé en termes de processus, la simulation sera alors basée sur

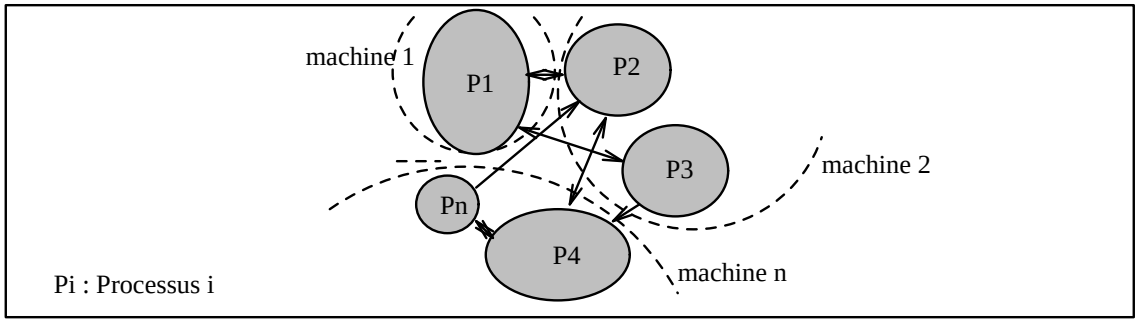

Figure 1.1: Les processus dans une simulation distribuée

la notion de processus. De plus dans un système distribué, chaque processus est indépendant et le contrôle de l'ensemble est distribué, c'est à dire qu'il se fait mutuellement entre les processus par l'intermédiaire des canaux de communication, sans contrôle global. Le simulateur fonctionne ainsi de façon asynchrone et la situation de chaque processus dans le temps sera déterminée à l'aide des messages ou clients. Un message comporte, en plus du type d'événement et autres informations, la date lui correspondant.

1.2.1 Caractéristiques

Outre sa rapidité potentielle[2] par rapport à la simulation séquentielle classique, elle a les caractéristiques suivantes, dues essentiellement à la coopération entre plusieurs machines :

- La simulation peut être adaptée à la configuration matérielle disponible. Par exemple, si on dispose de quatre machines pour une simulation comportant six processus, deux des machines simuleront chacune deux processus. Le placement des processus sur chaque machine est fonction de ces processus et de l'interaction entre eux (degré de communication). Deux processus fortement communicants peuvent par exemple être simulés par la même machine pour éviter les pertes de temps en communication.
- Chaque processus est totalement indépendant et la communication entre eux s'effectue sur les liens physiques entre les machines. Le contrôle de l'ensemble se fait de manière distribuée.
- Grâce à cette indépendance, chaque processus a ses données locales non partagées avec les autres.

1.2.2 Structure et fonctionnement général

Le simulateur inclut deux autres processus en plus de ceux dérivant de la structure du système physique à simuler : la source et l'éliminateur. Le premier est chargé de générer des clients (correspondant aux données entrant pour être traitées dans le système physique), de manière aléatoire et de les envoyer successivement dans le réseau, et le second de "ramasser" ceux qui sont traités (données sortant du système physique) et de les éliminer.

La figure 1.2 montre un exemple simple que nous allons faire fonctionner pour montrer le déroulement interne de la simulation. Supposons que le système a les données suivantes :

- Simulation entre les dates 0 et 20.
- La source génère les clients Cl_1, Cl_2 et Cl_3 respectivement aux dates 1, 10 et 20.
- Un client porte sa date de génération et sa date de simulation.

[2] toutefois fortement dépendante du degré de parallélisme intrinséque du système à simuler

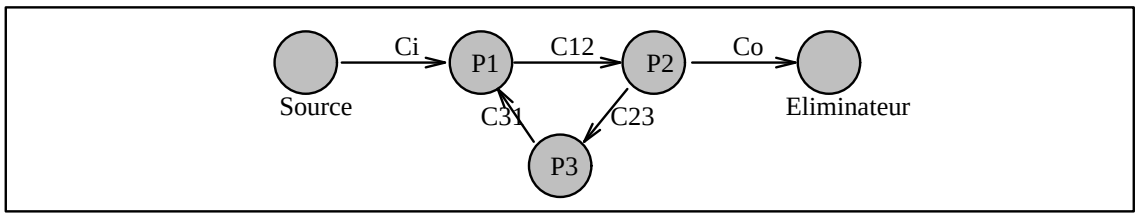

Figure 1.2: Exemple de structure d'un simulateur

- Un nouveau client doit passer une fois par chacun des trois processus que nous appelerons serveurs, avant de sortir du système. Son chemin sera donc P_1, P_2, P_3, P_1, P_2.
- P_1 a pour temps de service 3, P_2, 1 et P_3, 4.
- Les événements sont ARRIVEE et DEPART d'un client.

Le serveur P_1 attend sur ses canaux d'entrée, en l'occurrence C_i et C_{31}, des clients. Ceux venant de C_i étant à traiter et ceux venant de C_{31} à renvoyer. A la date 1, le client Cl_1 arrive sur C_i. P_1 va alors mettre dans son échéancier local l'événement ARRIVEE DE Cl_1 A LA DATE 1, simuler cet événement car c'est le premier, calculer la date de fin de service, ranger l'événement DEPART DE Cl_1 A LA DATE 4 (3+1) et passer à l'événement suivant, c'est à dire DEPART DE Cl_1 A LA DATE 4. Il va le simuler, envoyer le client Cl_1 muni de la date de simulation 4 à P_2 et recommencer son cycle. Ce client sera de retour à la date 9. Il sera alors simulé et renvoyé vers P_2. Un nouveau client Cl_2 arrive à la date 10 et P_1 recommence pour Cl_2 les mêmes traitements que pour Cl_1.

Les deux autres serveurs fonctionnent de la même manière : P_3 reçoit un client sur C_{23}, le traite et l'envoie sur C_{31}. P_2 reçoit un client sur C_{12}, le traite et, si c'est un nouveau client, l'envoie sur C_{23}, sinon l'envoie sur C_o. Au dernier passage du dernier client Cl_3 portant la date de génération 20, chaque serveur ainsi que la source et l'éliminateur vont s'arrêter, car ceci indique la fin de la simulation.

Nous avons simplifié le problème dans cet exemple pour bien montrer le fonctionnement global. Dans le cas réel, plusieurs autres problèmes peuvent apparaître. Prenons un exemple : le client Cl_2 portant la date 10 arrive sur P_1 avant Cl_2 portant la date 9. Il sera alors simulé avant Cl_1. Ce problème très courant est dû au fait que les différents processus s'exécutent de façon asynchrone. Il existe différentes méthodes pour la résolution de ce problème. Nous allons voir dans la partie qui suit la méthode dite "conservative".

1.2.3 La méthode de synchronisation conservative

(D'après [4])

1.2.3.1 Principe

Nous avons vu dans l'exemple précédent que la date de simulation dans un processus avance selon les messages traités (pour P_1 : 1, puis 10, puis 20). Si on avait utilisé la méthode conservative dans cet exemple, après avoir envoyé le premier client Cl_1 vers P_2, P_1 n'aurait pas traité le client Cl_2 de date 10 venant de la source avant Cl_1 de date 9 venant de P_3. Il aurait attendu que Cl_1 arrive sur le canal C_{31} avant de traiter celui venant du canal C_i (en l'occurrence Cl_2).

Le principe de cette méthode est donc le suivant : **la simulation ne peut être avancée à une date t que lorsque tous les clients de date inférieure à t ont été traités**. Autrement dit, si un client Cl_1 de date t_1 arrive sur un canal C_1, le processus attendra un client sur chacun de ses canaux d'entrée autres que C_1. Si sur ces canaux les clients sont tous de date supérieure à t_1, alors il traitera Cl_1, sinon il traitera celui de date minimale, soit par exemple celui venant du canal C_2. Après le traitement de ce dernier, il se peut que sur le même canal il y ait encore un autre client de date inférieure à t_1. Le processus l'attendra alors, le traitera et ne commencera le traitement de Cl_1 que lorsque tous les messages venant des autres canaux, autres que C_1, sont tous de date supérieure à t_1. En toute généralité, le principe s'applique aux événements et non aux clients. Un événement de date T ne peut être simulé que lorsque tous ceux de date inférieure l'ont été.

On s'aperçoit tout de suite qu'un processus peut se bloquer longtemps en attendant un message sur un canal (attente d'un événement "*arrivée*"). Ceci amène facilement à un deadlock si deux ou plus d'entre eux s'attendent mutuellement. L'algorithme dit *des messages nuls* [4], en particulier, permet de résoudre ces éventuels deadlocks.

1.2.3.2 Les processus logiques

On décompose le système à simuler en un ensemble de processus appelés processus physiques PP, communiquant par des messages. Un message envoyé par un PP à l'instant t est fonction des messages qu'il a reçus auparavant, et de son état courant. Ceci est la propriété de *réalisabilité* des seuls systèmes physiques que nous considérons ici. On définit le simulateur comme un ensemble de processus logiques LP. Chaque LP simule un PP du système physique. Son comportement dépend donc totalement du PP qu'il simule. Il y a un lien de communication, noté (i, j), de LP_i vers LP_j si PP_i envoie des messages vers PP_j. Le simulateur a un fonctionnement asynchrone grâce à l'indépendance des LP. Le contrôle de la progression se fait par les messages que nous appelons aussi *clients*, qui portent des dates. Ainsi un message m, envoyé par PP_i vers PP_j à la date t, est simulé par l'envoi par LP_i vers LP_j du doublet (t, m), ceci sur le canal (i, j). Si [(t_1, m_1) (t_2,m_2) ... (t_k, m_k)] sont les messages transmis de LP_i à LP_j, du début de la simulation à la date t, alors $0 \leq t_1 < t_2 < ... < t_k$.

1.2.3.3 La prédiction sur les sorties

Cette prédiction sur les sorties est un aspect fondamental de la méthode conservative car la résolution des deadlocks repose sur elle. Supposons que la date de simulation d'un LP_i est T_i, c'est à dire que le client en cours de traitement, venu par le canal (k, i) porte cette date. Il est alors possible pour LP_i de déduire la liste des messages qu'il enverra vers LP_j jusqu'à la date T_i', en fonction de ce qu'il a reçu auparavant jusqu'à la date T_i, indépendamment des messages qui lui parviendront entre les dates T_i et T_i'. On appelle PREVISION (*lookahead*) sur le canal (i, j) la valeur

$$L_{ij} = T_i' - T_i.$$

De façon plus formelle, la date jusqu'à laquelle LP_i peut prédire ce qu'il enverra vers LP_j est :

$$T_{OUTij} = T_{INi} + L_{ij},$$

où L_{ij} est la prévision, et $T_{INi} = \min_k T_{ki}$, c'est à dire le minimum des dates des canaux d'entrée de LP_i. T_{ki} étant la date du dernier message reçu sur le canal (k, i). Nous appellerons *prédiction* la date T_{OUTij}.

Cette définition de L_{ij} nous amène à la propriété de *prédictibilité*. Supposons que le système contient des processus reliés en boucle. Selon la définition de la réalisabilité que nous avons vue plus haut, un message donné peut être fonction de lui même dans cette boucle. En effet, un message envoyé par un PP_i à la date t est fonction de ce qu'il a reçu aux dates inférieures à t mais aussi à la date t. Pour éviter ce genre de situation, nous nous limitons ici aux systèmes qui vérifient la propriété de prédictibilité suivante : dans une boucle de processus, il y a au moins un canal (i, j) pour lequel la prévision L_{ij} est supérieure à E_p. $E_p \neq 0$ étant fixé comme le temps minimum entre l'envoi de deux messages successifs par un LP (c'est à dire qu'un LP n'envoie jamais deux messages de même date).

1.2.3.4 Les étapes

Au début de la simulation, les variables locales de chaque LP sont initialisées de la manière suivante :

- $T_i = 0$ (date courante du LP_i),
- $T_{ki} = (0, NUL)$ (Date de chaque canal d'entrée (k, i). Nous verrons la signification de NUL plus loin),
- $T_{ij} = (0, NUL)$ (Date de chaque canal de sortie (i, j)),
- A chaque instant, d'autres variables locales contiennent les futurs messages à envoyer.

Après son initialisation, un LP boucle sur les trois étapes suivantes jusqu'à la fin de la simulation : sélection, traitement et opérations d'entrées/sorties.

`SÉLECTION`

Durant cette étape, LP_i va sélectionner les canaux d'entrée et de sortie sur lesquels les opérations suivantes vont être effectuées. Nous avons vu plus haut qu'à la date t, un LP_i peut définir les messages qu'il a à envoyer sur le canal (i, j) jusqu'à la date T_{OUTij}, sans attendre d'autres messages si T_{OUTij} est supérieure à la date T_i de LP_i. On dit que le canal (i, j) est *ouvert* dans ce cas.

Soit $T_i = \min_k(T_{ij}, T_{ki})$, la date du LP_i. Les canaux sélectionnés ici sont les canaux d'entrée et les canaux de sortie ouverts, qui ont leur date égale à T_i. L'ensemble de ces canaux peut être défini par :

$$Next_i = \{k/\ T_{ki} = T_i\} \cup \{j/\ T_{ij} = T_i\ et\ T_{OUTij} > T_{ij}\}.$$

TRAITEMENT

Les traitements et la préparation des messages à envoyer sur les canaux de sortie sélectionnés, sont effectués dans cette étape. Soit un canal (i, j) où j appartient à $Next_i$. Entre la date courante T_{ij} et la date T_{OUTij}, deux cas peuvent se présenter :

Cas 1 : Il y a un ou plusieurs messages à envoyer sur le canal (i, j). Leurs dates calculées seront alors comprises dans l'intervalle $]T_{ij}, T_{OUTij}]$.

Cas 2 : Il n'y a aucun message à envoyer sur le canal (i, j). Dans ce cas, LP_i va préparer un message de synchronisation, appelé *message nul*, à envoyer sur ce canal. Ce message portera la date T_{OUTij} et indiquera à LP_j que LP_i ne lui enverra aucun message avant la date T_{OUTij}. LP_j saura alors qu'il n'a pas à attendre un message de date inférieure à T_{OUTij} sur le canal (i, j).

C'est ce mécanisme d'envoi de messages nuls, implanté dans chaque LP, qui permet d'éviter les deadlocks dans cette méthode. Il n'y aura jamais d'attente infinie car chaque LP signalera à son successeur l'absence de message lui provenant.

ENTRÉES/SORTIES

Après l'étape de traitement, LP_i exécute en parallèle l'envoi sur les canaux ouverts des messages préparés, et l'attente de nouveaux messages sur les canaux d'entrée sélectionnés dans $Next_i$. Après une opération sur un canal (entrée ou sortie), la date de ce canal est avancée à la date du dernier message passé sur ce canal (reçu ou envoyé). La date T_i de LP_i est également mise à jour dans cette étape.

Soit Z la date finale de la simulation. Tant que la date T_i de LP_i n'atteint pas Z, LP_i bouclera sans arrêt sur les trois étapes précédentes. T_i sera égale à Z quand tous les canaux d'entrée de LP_i atteindront cette date. En effet sur un canal portant la date Z, il n'y aura plus d'autres messages car l'expéditeur l'a déjà fermé après l'envoi du dernier client (Z, m) ou (Z, NUL).

Durant la phase de traitement, LP_i peut générer un message de date égale à Z. Il l'enverra sur le canal approprié (i, j), indiquant à LP_j par la même occasion qu'il ferme le canal (i, j). Il peut également générer un message de date supérieure à Z, auquel cas il enverra sur (i, j) le message (Z, NUL) qui indique de la même façon que le canal (i, j) est fermé.

Différentes variantes intéressantes de cette méthode ont été proposées par différents auteurs [8][9]. Nous n'en n'exposons aucune dans ce rapport, bien que PARSEVAL en exploite certaines.

Chapitre 2

Le simulateur PARSEVAL

PARSEVAL, un simulateur distribué expérimental de réseaux de files d'attente, fonctionnant sur un réseau de transputers, est une implémentation d'une version légèrement améliorée de la méthode conservative décrite dans le chapitre précédent. En effet, il utilise des tampons "intelligents" permettant de limiter la quantité de messages nuls circulant dans le système. Avant la description de PARSEVAL, un petit rappel sur les paramètres de définition d'une file d'attente permet de bien cadrer les systèmes à simuler.

2.1 Les réseaux de files d'attente

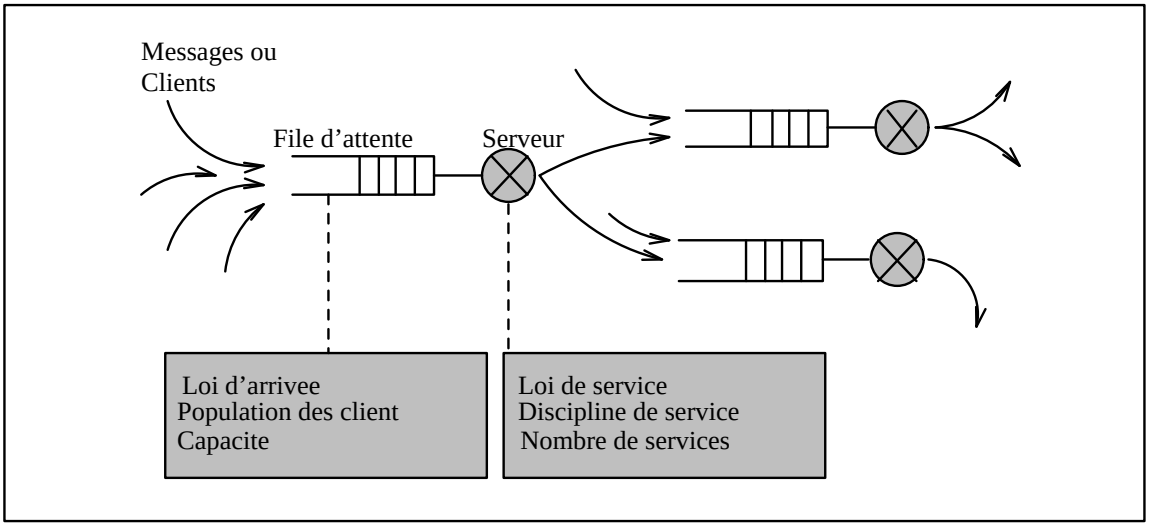

Figure 2.1: Réseau de files d'attente

Un ensemble de personnes faisant la queue devant un guichet de ventes de billets est un exemple de file d'attente. Le guichet est alors le serveur de la file et "traite" les clients un à un.

Une file d'attente, notée $A/B/C/K/m/Z$, est définie par six paramètres:

- A : Processus d'entrée. C'est la loi de distribution des intervalles entre les arrivées de clients à l'entrée de la file.
- B : Processus de service. C'est la loi qui régit le service fourni par le serveur de la file. Pour A et B, différentes notations peuvent être utilisées selon le processus, par exemple :
  - GI : loi générale, avec indépendance des variables successives.
  - G : loi générale.

  - M : loi exponentielle.
  - D : loi constante.

- C : Nombre de serveurs associés à la file.
- K : Capacité maximale. C'est le nombre maximal de clients pouvant se mettre en attente dans la file.
- m : Population des usagers. C'est le nombre maximal de clients pouvant passer dans la file.
- Z : Discipline de service. C'est la politique de gestion des clients dans la file. Il existe différentes façons de gérer les clients, nous allons prendre quelques exemples :
  - $FCFS\,(First\,Come\,First\,Served)$: Le premier client arrivé est le premier servi.
  - $LCFS\,(Last\,Come\,First\,Served)$: Le dernier client arrivé est le premier servi.
  - $QUANTUM$: Les clients sont servis à tour de rôle par tranches de temps de durée fixe. Cette durée est appelé "*quantum*".
  - $PRIORITE$: Les clients sont classés par ordre de priorité de service.
  - $PS\,(Processor\,Sharing)$: Tous les clients de la file sont en service, ils se partagent le serveur.

Les lois de service et les politiques de gestion de la file implémentées dans PARSEVAL sont plus détaillées dans la partie "Simulateur d'une file d'attente ". Lorsque les trois derniers éléments de la notation ne sont pas précisés, il est sous-entendu que $Z = FCFS$, $m = +\infty$ et que $K = +\infty$.

Exemple : La file $M/D/2/2/6/FCFS$ a des intervalles entre arrivées distribués de manière exponentielle, une loi de service constante et 2 serveurs. Elle est de taille 2, accepte le passage de 6 clients et est gérée selon la politique $FCFS$.

De tels objets élémentaires sont alors combinés pour former des réseaux de files d'attente, permettant par exemple de modéliser les partages de ressources dans un système informatique. Il y a donc échange de clients entre les différentes files d'attente composant le réseau (figure 2.1). Les *prédécesseurs* d'une file sont celles qui lui envoient des clients, ses *successeurs* sont celles à qui elle envoie des clients.

2.2 Le système PARSEVAL

Il est composé de deux parties distinctes : une interface graphique qui s'exécute sur l'ordinateur hôte (sous SunTools sur station Sun), et un système de simulation qui s'exécute sur un réseau de transputers. L'interface graphique permet à l'utilisateur de décrire le modèle de réseau de files d'attente, et de générer un code qui sera par la suite exécuté par le système de simulation.

Ce système de simulation (figure 2.2) est constitué d'un réseau de routage qui joue le rôle d'aiguilleur de messages, et d'un ensemble d' "*applications*" qui simulent les files d'attente. Dans la suite de ce rapport, "*simulateur*" indiquera "*simulateur d'une file d'attente munie de son serveur*". Le système de routage implanté dans PARSEVAL est un réseau en anneau[1], dont le nombre de noeuds peut être défini selon le matériel disponible et le réseau de files d'attente à simuler, le but étant de partitionner la simulation sur un réseau de transputers. A chaque noeud, appelé "*routeur*", sont associés des simulateurs ou des sources de clients, et l'ensemble routeur-simulateurs/sources peut être placé sur un processeur. L'interface graphique et les différentes parties du système de simulation sont décrits dans les paragraphes qui suivent.

[1] dans la première version PARSEVAL 1.0

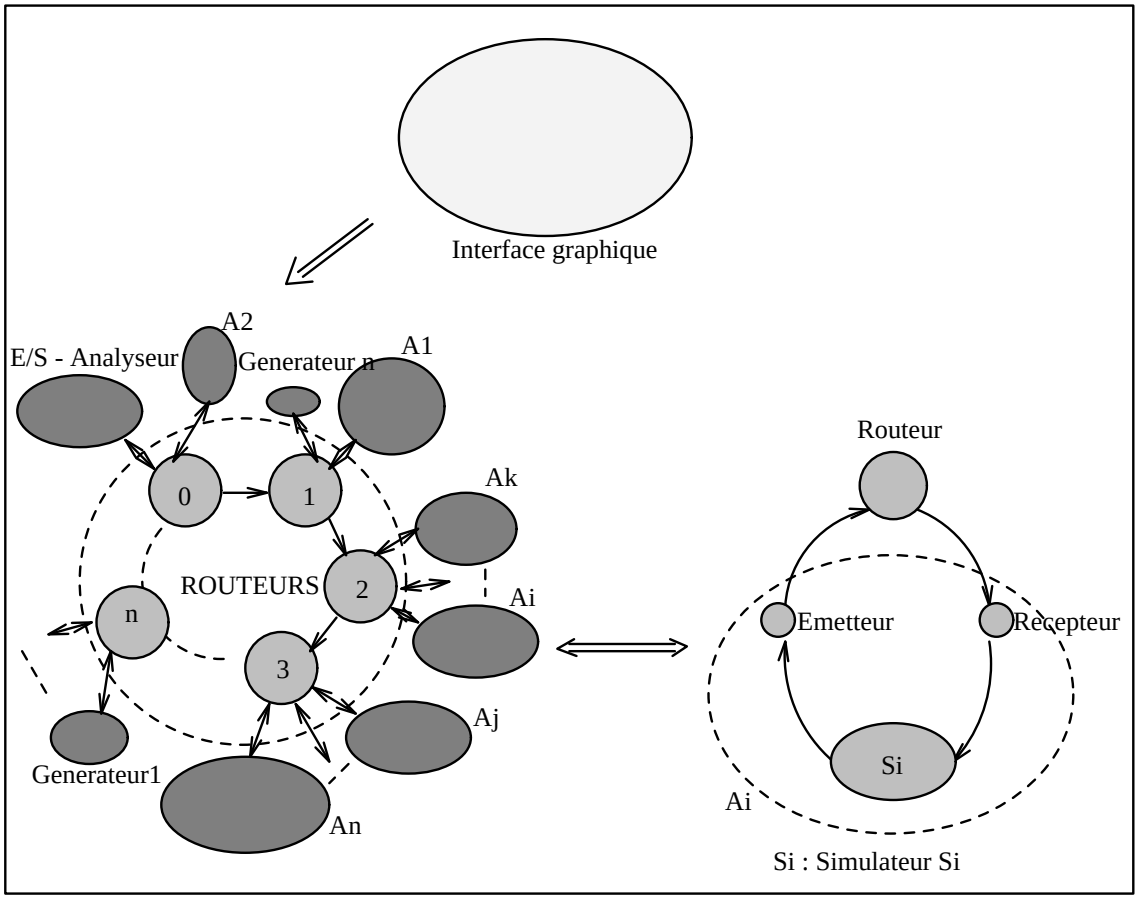

Figure 2.2: Structure globale de PARSEVAL

2.2.1 L'interface graphique

2.2.1.1 GSS

GSS[2] est un outil de description graphique de réseaux de files d'attente. La bibliothèque GSS fournit un ensemble de fonctions et de procédures écrites en langage C, permettant à l'utilisateur d'effectuer différentes opérations notamment la saisie interactive et la sauvegarde d'un réseau, ou le chargement en mémoire d'un modèle déjà stocké dans un fichier, un modèle chargé en mémoire étant une structure de données composée d'objets tels que les nœuds et les liens d'un réseau de files d'attente.

Les différents paramètres de la structure de données sont accessibles et peuvent être manipulés à l'aide des fonctions et des procédures de la bibliothèque GSS. L'utilisateur peut ainsi effectuer une multitude de traitements, par exemple dans notre cas la génération de code en langage OCCAM [16] destiné à la simulation du modèle.

La forme et le contenu de la structure de données où seront rangés les modèles saisis sont décrits à l'avance par l'utilisateur à l'aide d'un langage spécifique de GSS, et rangés dans un fichier "*fic.gdl*". Cette description porte sur les différents objets c'est à dire le graphe, les nœuds, les liens et les modules statistiques. Un graphe est composé de nœuds, liens et modules statistiques, et forme une structure complète pour un modèle. Les attributs (loi de service, politique de gestion, etc...) et leurs types ainsi que les paramètres graphiques, associés à chacun des objets, doivent être spécifiés.

[2]GSS a été dévelopé dans le cadre du projet ESPRIT II IMSE (Integrated Modelling System Environment) *The IMSE project is a collaborative research project supported by the CEC as ESPRIT project no 2143. It is being carried out by the following organisations: STC Technology Ltd, Thomson CSF, Simulog A.S., University of Edinburgh, INRIA, IPK (Berlin), University of Dortmund, University of Pavia, SINTEF (University of Trondheim), University of Turin and University of Milan.*

2.2.1.2 La génération de code

L'interface graphique du système est un programme écrit en C et effectue les opérations suivantes.

- Initialisation de GSS.
- Chargement du type de graphe c'est à dire de la structure de données décrite par l'utilisateur.
- Création d'une fenêtre pour la saisie interactive. Un menu standard associé à cette fenêtre permet la saisie graphique, la sauvegarde et le chargement d'un modèle. L'interface ajoute un choix supplémentaire dans ce menu standard pour permettre à l'utilisateur d'exécuter la procedure de génération de code proprement dite. Cette procédure accède aux différents paramètres de la structure de données ainsi qu'aux liens entre les objets, et génère le code OCCAM correspondant pour la simulation.

Un exemple de représentation graphique d'un modèle est montré par la figure 2.3 et le code OCCAM généré est le suivant :

```
VAL tdeb IS 0.000000 (REAL64):
VAL tmax IS 1000.000000 (REAL64):                    --Durée de la simulation

VAL nb.processeur IS 4:                              --Nombre de processeurs

CHAN OF SP fs, ts:                                   --Les canaux physiques
[nb.processeur-2]CHAN OF message ring:
CHAN OF message mux.ana, ana.mux:
CHAN OF message app.es.ana, ana.app.es:

-- station1 --                                       --Paramètres de la station 1
VAL nb.pred1 IS 1:
VAL pred1 IS [2]:                                    --Son prédécesseur est le 2 (gene1)
VAL proba.pred1 IS [1.000000(REAL64)]:
VAL nb.succ1 IS 1:
VAL succ1 IS [no.trappe]:                            --Pas de successeur
VAL proba.succ1 IS [1.000000(REAL64)]:
VAL law1 IS loi.erl:                                 --Loi de service ERLANG
VAL lawparam1 IS [7.000000(REAL64),2.000000(REAL64)]:
VAL sched1 IS fifo:                                  --Politique de gestion FIFO
VAL schedparam1 IS [0.0(REAL64)]:
VAL nb.serv1 IS 2:                                   --Deux serveurs
VAL graine.aleatoire1 IS 6573(INT64):

-- gene1 --                                          --Paramètres du générateur
VAL nb.succ2 IS 1:
VAL succ2 IS [1]:
VAL proba.succ2 IS [1.000000(REAL64)]:
VAL law2 IS loi.exp:
VAL lawparam2 IS [13.000000(REAL64)]:
VAL nb.client.max2 IS 86:                            -- 86 clients à générer
VAL graine.aleatoire2 IS 23987(INT64):


PLACED PAR                                           --Placement sur les processeurs

  PROCESSOR 0 T8
    PLACE fs AT num.fs:
    PLACE ts AT num.ts:
    PLACE app.es.ana AT es.app.es.ana:
    PLACE ana.app.es AT es.ana.app.es:
    PAR
      app.es(0,ana.app.es,app.es.ana,fs,ts)          --Application d'E/S

  PROCESSOR 1 T8
    PLACE app.es.ana AT ana.app.es.ana:
    PLACE ana.app.es AT ana.ana.app.es:
    PLACE mux.ana AT ana.mux.ana:
    PLACE ana.mux AT ana.ana.mux:
    PAR
      analyseur(app.es.ana,ana.app.es,mux.ana,ana.mux)

  PROCESSOR 2 T8
    PLACE mux.ana AT mux.mux.ana:
    PLACE ana.mux AT mux.ana.mux:
    PLACE ring[0] AT left[0]:
    PLACE ring[1] AT right[0]:
    [2]CHAN OF message routeur.application:          --Liens internes entre routeur et applications
    [2]CHAN OF message application.routeur:
    [2]CHAN OF message init.application:
```

```
    [2]CHAN OF message special:
    PAR
      routeur(2,ring[1],ring[0],2,
              routeur.application,init.application,
              special,application.routeur)

      – app.es –
      mux.ext(routeur.application[0],init.application[0],
              special[0],application.routeur[0],
              ana.mux,mux.ana)

      – gene1 –
      generateur(2,routeur.application[1],init.application[1],
              application.routeur[1],special[1],
              law2,lawparam2,
              nb.succ2,succ2,proba.succ2,
              nb.client.max2,
              graine.aleatoire2,
              tdeb,tmax)


PROCESSOR 3 T8
    PLACE ring[1] AT left[1]:
    PLACE ring[0] AT right[1]:
    [1]CHAN OF message routeur.application:
    [1]CHAN OF message application.routeur:
    [1]CHAN OF message init.application:
    [1]CHAN OF message special:
    PAR
      routeur(3,ring[0],ring[1],1,
              routeur.application,init.application,
              special,application.routeur)

      – station1 –
      server(1,routeur.application[0],init.application[0],
              application.routeur[0],special[0],
              nb.serv1,
              law1,lawparam1,
              sched1,schedparam1,
              nb.pred1,pred1,proba.pred1,
              nb.succ1,succ1,proba.succ1,
              graine.aleatoire1,
              tdeb,tmax)
```

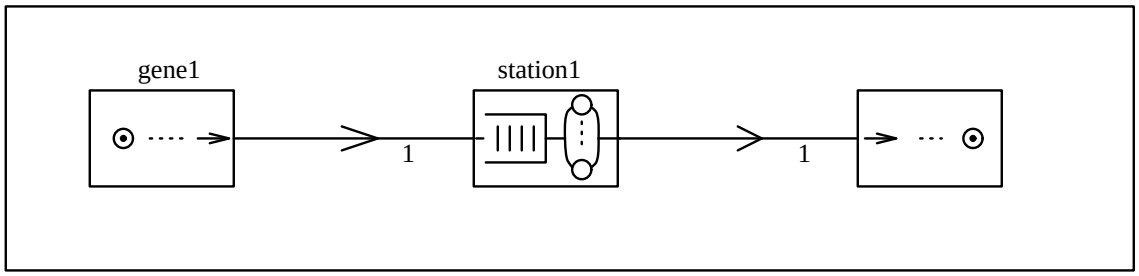

Figure 2.3: Représentation graphique dans GSS

Nous verrons dans le paragraphe qui suit les différentes procédures utilisées dans ce code qui servira de programme principal au système de simulation.

2.2.2 Le réseau de routage - Fonctionnement global

Le système a été conçu pour simuler un type quelconque de réseau de files d'attente. Le nombre de nœuds du système de routage est modifiable selon le réseau à simuler, et par conséquent le nombre de files d'attente peut augmenter "indéfiniment". D'une manière évidente, ceci peut complexifier l'ensemble, l'amener à saturation et compromettre ainsi ses performances. Des limites ont donc été imposées quant au nombre de routeurs et au nombre maximal de files d'attente.

Les messages circulant dans le système peuvent être classés en deux groupes.

- D'une part, les clients et les messages nuls qui sont les propres messages du réseau de files d'attente. Ils incluent le numéro de l'expéditeur, celui du destinataire et différentes informations telles que le numéro du client, sa date d'envoi etc.

- D'autre part, les jetons de synchronisation qui servent pour l'arrêt de PARSEVAL. Ces jetons sont utilisés pour synchroniser les arrêts normaux (date finale de simulation atteinte) et les arrêts par erreur (par exemple échéancier plein). Un simulateur qui veut s'arrêter envoie une demande à son routeur qui lui en donnera l'autorisation après s'être synchronisé avec tout le réseau.

Le routeur (procédure "*routeur*") dirige les messages qu'il reçoit directement vers le destinataire si c'est un de ses processus associés, ou vers son voisin du réseau si ce n'est pas le cas. Outre les routeurs, les autres processus constituant PARSEVAL sont : l'application d'entrée/sortie, l'analyseur qui se chargera de collecter des mesures effectuées sur le système pour sa surveillance, les sources de clients et les simulateurs qui sont chacun partagés en trois parties: l'émetteur, le récepteur et le simulateur proprement dit. L'émetteur et le récepteur sont des "*buffers intelligents*" qui permettent de limiter la quantité de messages nuls circulant, et d'alléger le travail du simulateur. C'est une amélioration de la méthode conservative décrite initialement dans [4].

2.2.3 L'application d'entrée/sortie

C'est le processus qui permet au système de simulation d'être en contact avec son environnement et avec l'utilisateur durant son exécution (procédure "*app.es*"). Il se charge de la gestion du clavier, donc des ordres de l'utilisateur et de l'affichage à l'écran (état du système, résultats etc.).

Dans l'état actuel de PARSEVAL, l'utilisateur peut, soit arrêter l'ensemble du système, soit arrêter l'application d'E/S, donc laisser le réseau fonctionner "tout seul", soit envoyer des messages du clavier vers l'écran via le réseau pour vérifier d'éventuels blocages.

2.2.4 L'analyseur

Un module de surveillance ou de "*monitoring*" est en cours de développement. Ce module effectuera des mesures de l'activité de chaque processeur et des mesures de la quantité de communications entre les processus et entre les processeurs durant l'exécution du système. Les mesures ainsi collectées seront envoyées à l'analyseur (procédure "*analyseur*") qui les stockera et les mettra en forme avant de les envoyer vers l'écran. Le rôle de la procédure "*mux.ext*" est simplement de multiplexer les quatre canaux du routeur vers l'application d'entrée/sortie via l'analyseur.

2.2.5 L'application source de clients

Elle joue le rôle du "*générateur*" (procédure de même nom), un des deux processus spéciaux décrits dans la partie "*Simulation distribuée*" du chapitre précédent. La source génère des clients selon une loi donnée et les envoie à ses successeurs, chaque successeur ayant une probabilité de recevoir un client. L'utilisateur doit particulièrement faire attention à la valeur à donner aux paramètres de ces lois car si une source génère des messages trop fréquemment, elle risque de saturer le réseau (saturation physique) et de le bloquer avant que la simulation n'arrive à son terme.

Le deuxième processus spécial dit "*éliminateur*" dans le chapitre précédent, n'existe pas réellement. Un client qui sort du réseau est simplement éliminé par le dernier simulateur par lequel il est passé.

2.2.6 L'émetteur

Les trois procédures "*émetteur*", "*récepteur*" et "*simulateur*" décrits par les paragraphes suivants forment la procédure "*server*" qui est le simulateur d'une file d'attente. L'émetteur permet au simulateur de ne pas se bloquer sur l'envoi d'un message pendant que le routeur est indisponible, en faisant office de tampon entre eux.

L'émetteur est composé de deux processus s'exécutant en parallèle et utilise une liste de rangement des clients en attente du routeur (figure 2.4).

Le *processus B* est chargé de la communication avec le réseau de routage, c'est à dire l'attente de la disponibilité du routeur et la réception du message d'arrêt lui provenant.

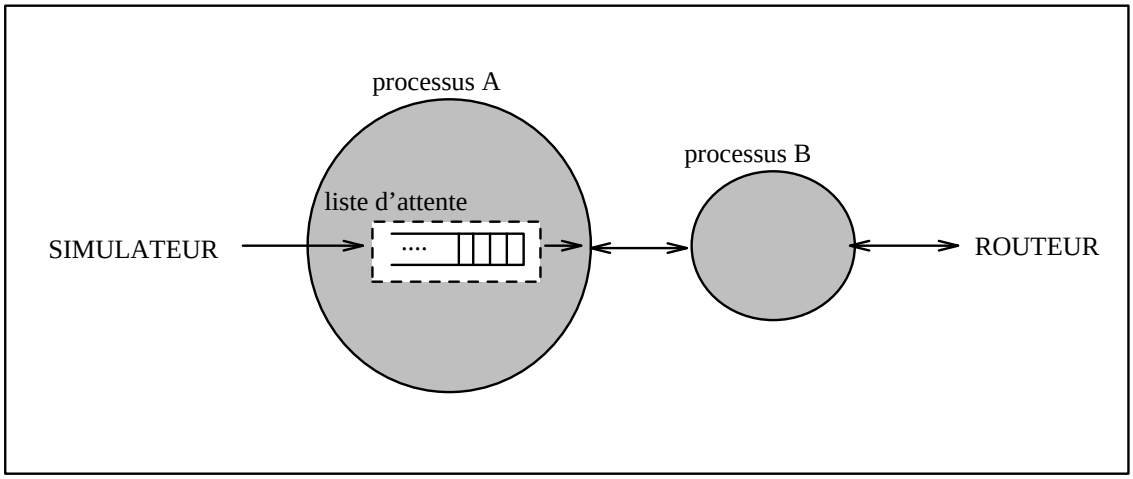

Figure 2.4: Structure de l'émetteur

Le *processus A* s'occupe de la communication avec le simulateur, du rangement des messages envoyés par celui-ci dans la liste d'attente, et de l'élimination des messages nuls inutiles. En effet, un message nul se trouvant encore dans la liste, et suivi d'un client ou d'un autre message nul, de même destination que lui, ne doit plus être envoyé, car son effet est directement annulé par le suivant. Ceci limite la quantité de messages nuls circulant dans le système.

Lorsque le routeur est prêt à recevoir un message, B lui envoie celui qu'il avait en attente, demande à A le suivant c'est à dire le premier de la liste, et attend à nouveau la disponibilité du routeur.

2.2.7 Le récepteur

Comme l'émetteur, il joue le rôle de tampon entre le routeur et le simulateur, et effectue en plus la sélection des clients à l'arrivée. Grâce à cette sélection, le récepteur n'envoie vers le simulateur que les clients de date d'arrivée minimale, c'est à dire les clients dont tous les prédécesseurs ont été envoyés.

Cette sélection constitue une partie de l'étape SÉLECTION et une partie de l'étape entrée/sortie de la méthode conservative décrite dans le chapitre précédent. En effet, elle ne concerne que les canaux d'entrée, la sélection sur les canaux de sortie étant faite dans le simulateur.

Le récepteur est un processus qui utilise deux listes de clients (figure 2.5) pour remplir ses rôles. Il stocke les clients reçus du routeur dans la *liste* 1, qui est un ensemble de sous-listes. Chacune d'elles correspond à une provenance, donc à un prédécesseur du simulateur associé au récepteur considéré. C'est dans cette *liste* 1 que s'effectue la sélection.

Les clients choisis sont rangés dans la *liste* 2 en attendant que l'application les demande. Ils sont classés et envoyés par ordre de date.

Pour illustrer le fonctionnement du récepteur, considérons un simulateur A dont les prédécesseurs sont C, D et F. La *liste* 1 est donc constituée par les sous-listes respectives SC, SD et SF. Au début de la simulation, ces sous-listes portent chacune la date 0. Supposons que les clients arrivent dans l'ordre suivant :

- *client* 1 : date 10 , provenance D, rangé dans SD,
- *client* 4 : date 9 , provenance C, rangé dans SC,
- *client* 7 : date 9.5, provenance C, rangé dans SC,
- *client* 5 : date 12 , provenance F, rangé dans SF,

Le *client* 4, puis le *client* 7 seront alors mis dans la *liste* 2, et la date de SC sera avancée à 9.5 . Cette opération peut être effectuée car aucun autre client de date inférieure à 9.5 ne peut arriver, l'ordre des clients de même provenance étant garanti. Par contre, 1 et 5 ne peuvent être rangés dans la *liste* 2 car un autre client de date inférieure à 10 peut venir de C.

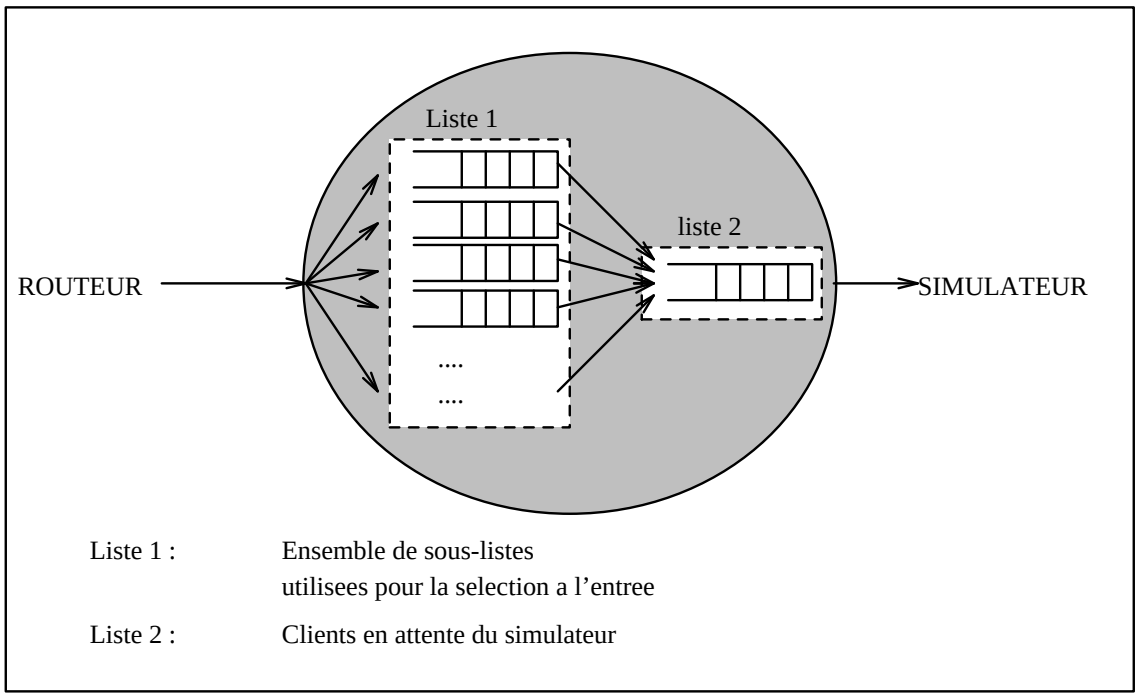

Figure 2.5: Structure du récepteur

2.2.7.1 Blocage

Supposons maintenant que le simulateur demande un client, puis un autre au récepteur. 4 et 7 lui seront alors envoyés successivement. Si le simulateur en demande encore, le récepteur lui enverra un message nul lui indiquant qu'il est bloqué en attente ($liste2\,vide$), et qu'il n'acceptera plus aucune demande, tant que la date du message suivant (nul ou client) qu'il enverra n'a pas augmenté. Autrement dit, le simulateur ne recevra rien tant que la $liste\,2$ est vide, et tant que la date minimale parmi celles des canaux d'entrée vides n'a pas augmenté. Cette date est celle portée par le message nul que le récepteur vient d'envoyer, en l'occurence 9.5 .

Ce blocage de transmission entre le simulateur et son récepteur permet au simulateur de limiter la quantité de messages nuls de prédiction qu'il envoie à ses successeurs. En effet, le simulateur envoie sa prédiction à chaque fois qu'il reçoit un nul de son récepteur, c'est à dire à chaque début de blocage à l'entrée comme nous venons de le voir, mais aussi à chaque déblocage par un message nul, que nous verrons dans la suite de cet exemple. Il est inutile d'envoyer un autre nul de prédiction si la date qu'il porte n'a pas augmenté.

2.2.7.2 Déblocage par un client

Si, dans l'état actuel de notre exemple, le récepteur reçoit un nul de date 13 provenant de C, la date de SC sera avancée à 13, le $client\,1$ transféré dans la $liste\,2$ et la date de SD avancée à 10. Ce nul indique que C n'enverra aucun client de date inférieure à 13. A la demande du simulateur, le $client\,1$ lui sera envoyé, suivi d'un nul de date 10. Et il y a à nouveau blocage de transmission.

2.2.7.3 Déblocage par un message nul

Après ce nouveau blocage si un message nul de date 11 arrive de D, la date de SD sera avancée à 11. Comme aucun client ne peut être rangé dans la $liste\,2$, c'est un message nul de date 11 qui sera envoyé au simulateur à sa prochaine demande, et l'ensemble récepteur-application se bloque à nouveau sur l'entrée.

2.2.8 Le simulateur d'une file d'attente

Comme l'indique le titre, c'est le processus qui simule une file d'attente munie de son serveur. Sa position dans le modèle, c'est à dire ses prédécesseurs et ses successeurs avec les probabilités associés lui est passée en paramètre. Les types d'événement considérés sont :

- arr : arrivée,
- ds : début de service,
- fq : fin de quantum (dans le cas où la politique de gestion est un partage de temps en tranche, ou quantum),
- fs : fin de service.

Dans la suite, nous abrègerons "événement de type arrivée" par **arr**, "événement de type début de service" par **ds**, "événement de type fin de quantum" par **fq** et "événement de type fin de service" par **fs**.

2.2.8.1 Les paramètres de définition

Différents types de file d'attente peuvent être simulés. Trois des paramètres définissant un type donné sont transmis au simulateur à son appel :

- le nombre de serveurs de la file d'attente,
- la loi de service et ses paramètres, et
- la politique de gestion de la file et ses paramètres.

La valeur de la **capacité maximale**, définie en variable globale et accessible à tous les simulateurs, est identique pour toutes les files d'attente, et l'atteinte de cette valeur provoque une situation d'erreur. Ceci est dû au fait que'il est impossible de dire lorsqu'une file est pleine, si le système simulé est en surcharge ou s'il s'agit simplement d'un "*pic*" de la variable d'état "*longueur de la file*"à cet instant.

Le nombre total de clients passant dans un simulateur est illimité durant le fonctionnement de PARSEVAL. La seule limitation possible de la population est le remplissage d'un des buffers du système, dû au nombre trop important de messages en circulation. Dans ce cas, le simulateur s'arrête sur un cas d'erreur.

Le sixième paramètre caractéristique, **la loi d'arrivée**, est défini par les liens qui relient les différentes files d'attente dans le système. Comme nous l'avons vu plus haut, ces liens sont passés en paramètres au simulateur.

2.2.8.2 La structure

Comme le montre la figure 2.6, le simulateur reçoit les messages du récepteur et les renvoie vers l'émetteur. Il utilise les trois listes ci-après pour la gestion des clients.

- Une file d'attente qui est une liste chaînée de clients, classée par ordre croissant de date d'arrivée.
- Un échéancier qui est une liste chaînée d'événements, classée par ordre croissant de date. Un événement de date T, y est inséré après ceux de date inférieure à T et tous ceux de même date.
- Un serveur qui est une liste des clients en service, ayant une taille limitée au nombre maximal de services offerts.

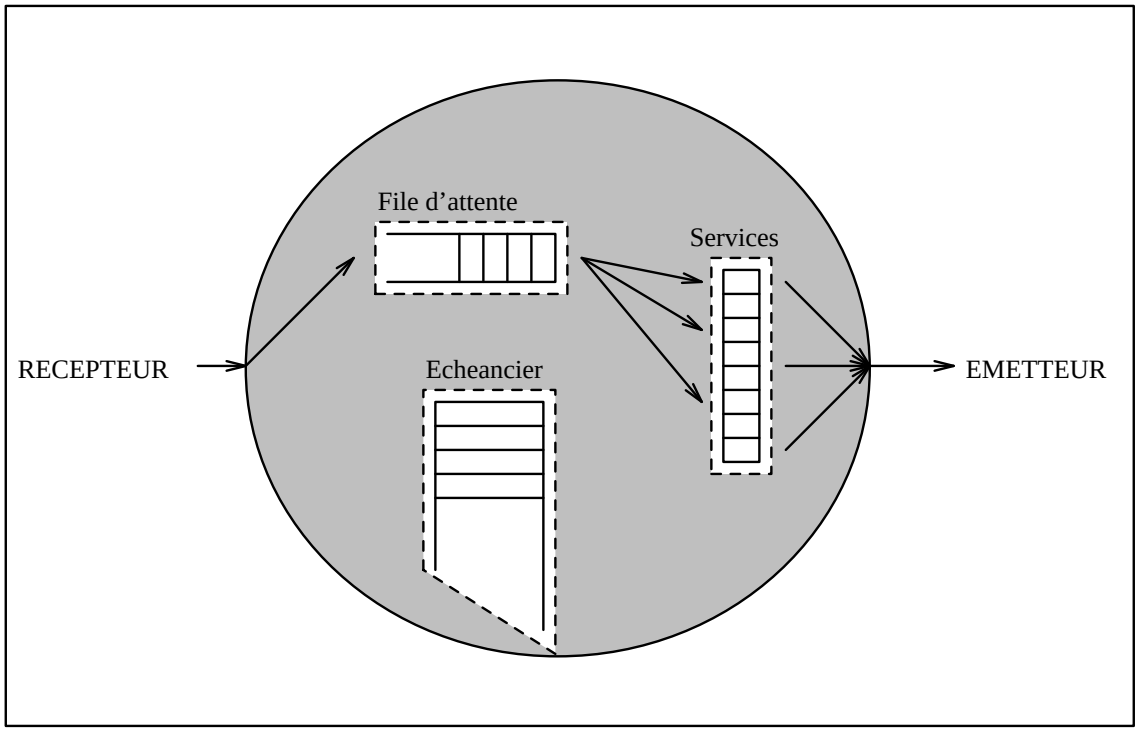

Figure 2.6: Structure d'un simulateur

2.2.8.3 La génération d'événements

- Un **arr** est créé à la réception d'un client issu du récepteur.
- Le traitement d'une **préemption**, d'un **fq** ou d'un **fs**, génère un **ds**. Selon la politique de gestion et le nombre de serveurs disponibles, la simulation d'un **arr** peut également générer un **ds**.
- Un **fs** ou un **fq** est créé après le traitement d'un **ds**.

La figure 2.7 récapitule la génération et le rangement des événements dans l'échéancier. Un autre événement appelé **arr non valide** est généré lorsque le simulateur reçoit un message nul. Son rôle est expliqué plus loin dans le paragraphe "Les messages nuls - La prédiction sur les sorties".

2.2.8.4 Le fonctionnement du simulateur

La simulation est dirigée par les événements. Le premier élément de l'échéancier n'est simulé que si tous ceux qui le précèdent dans le temps l'ont été. Autrement dit, le simulateur traite un événement de date t, seulement s'il est sûr que la réception suivante d'un client, ne peut générer un **arr** de date inférieure à t; les autres événements de l'échéancier étant tous de date supérieure à t. Le fonctionnement du simulateur est montré par la figure 2.8.

L'algorithme global implémenté dans PARSEVAL est le suivant :

Tant que *non fin*
attendre un message à l'entrée
Si *réception d'un client*
alors
génération d'un arr
sinon (réception d'un message nul)
génération d'un arr non valide

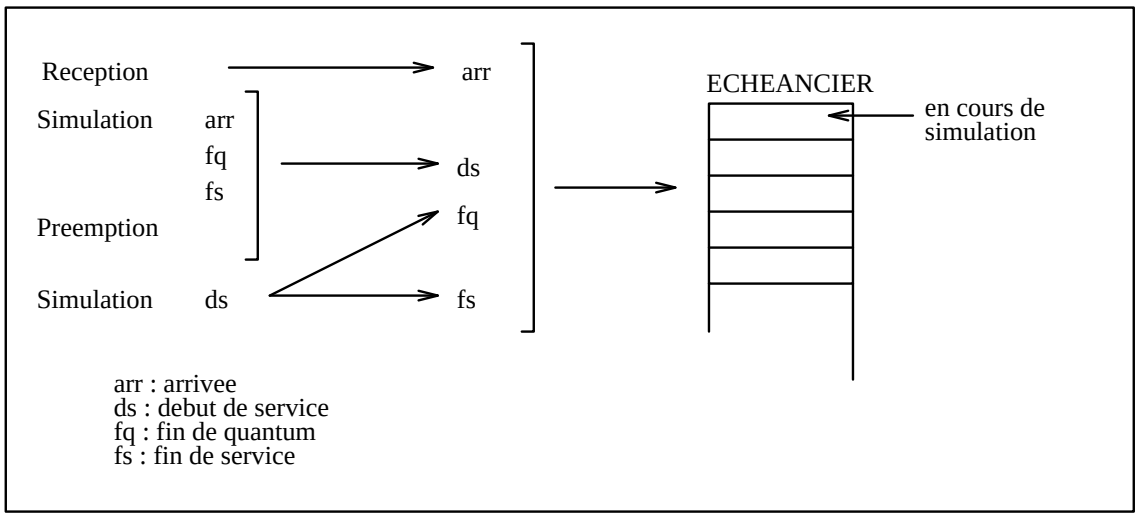

Figure 2.7: Génération d'événements

fin si
Faire
Si *premier événement* $\neq$ *arr non valide*
alors
simuler l'événement
sinon
calculer la prédiction
Si *date de prédiction* $>$ *date dernier nul envoyé*
alors
envoyer le message nul de prédiction
fin si
fin si
jusqu'à *événement* $=$ (*arr* **OU** *arr non valide*)
Simuler tous les événements de même date que ce arr
fin tant que

Nous allons dérouler cet algorithme pour bien éclairer le fonctionnement du simulateur. Au début de la simulation, l'échéancier est vide, le simulateur se bloque en attente d'un client à son entrée. Dès que celui-ci arrive, un **arr** est généré, rangé au début de l'échéancier et simulé tout de suite, car aucun autre client de date inférieure ne peut arriver (les clients envoyés par le récepteur sont ordonnés dans le temps). Ce traitement va créer un **ds** au début de l'échéancier après l'enlèvement du **arr**. Ce **ds** porte la même date que le **arr** précédant, car tous les serveurs sont disponibles. Il sera donc simulé immédiatement, créant ainsi un **fs**. Ce dernier n'ayant pas la même date, le simulateur sera obligé d'attendre à nouveau à son entrée l'arrivée d'un client ou d'un message nul. Deux cas peuvent alors se présenter.

Cas 1: La date d'arrivée du client reçu est inférieure à celle du dernier **fs**, alors le nouveau **arr** sera inséré au début de l'échéancier et simulé tout de suite. Il en sera de même pour les autres événements générés, qui portent la même date. Ensuite, le simulateur se bloquera à nouveau sur l'événement suivant portant une date supérieure.

Cas 2: La date d'arrivée du client reçu est, supérieure ou égale à celle du **fs**. Ce dernier sera alors simulé, ainsi que tous les autres événements de date inférieure au nouveau **arr**, générés par ce traitement. Le **arr** sera traité par la suite, ainsi que ceux qui ont la même date, avant le nouveau blocage.

Un simulateur s'arrête dès qu'il a rencontré un événement de date supérieure ou égale à la date

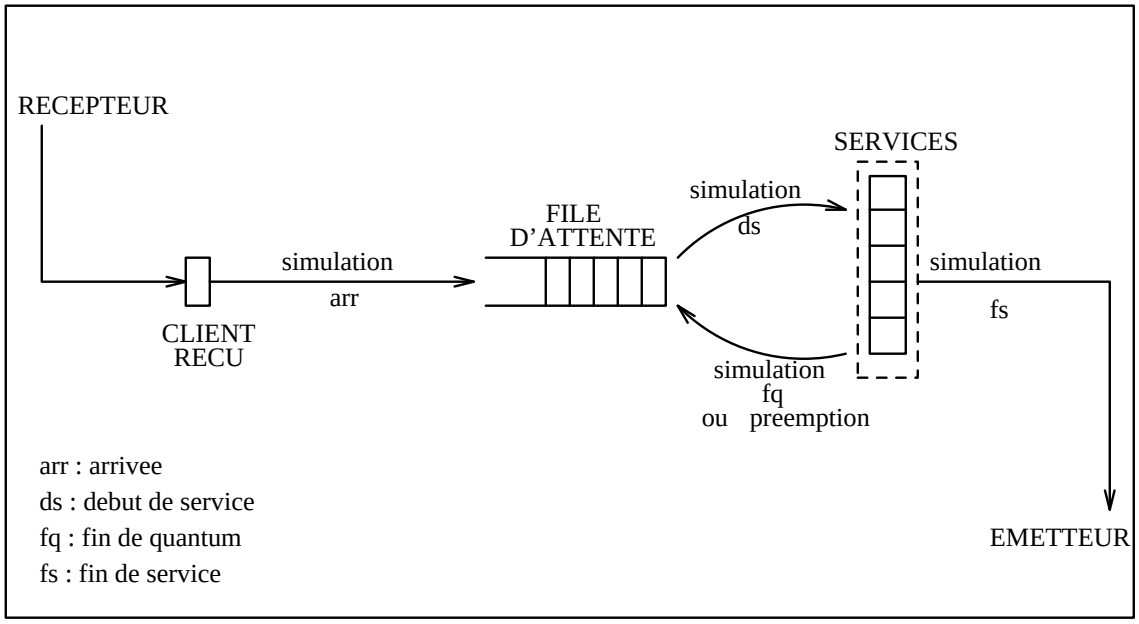

Figure 2.8: Fonctionnement d'un simulateur

finale. Cette date finale lui est indiquée en paramètre. Le fonctionnement normal d'un simulateur, sans réception ni envoi de message nul de prédiction, est ainsi décrit. La simulation de chacun des événements est décrite plus loin.

2.2.8.5 Les messages nuls - La prédiction sur les sorties

Tant que le simulateur ne se bloque pas en attente de client à son entrée, il n'envoie aucun message nul. En effet, il est inutile d'émettre une prédiction, alors qu'un client normal peut être envoyé immédiatement. Rappelons que **la prédiction** est la date maximale avant laquelle le simulateur n'enverra aucun client, et **la prévision** (*lookahead*) est l'intervalle de temps entre la date du message nul reçu et la prédiction.

Après réception d'un message nul issu du récepteur, le simulateur génère un événement *arr non valide* de même date que le nul, et continue à simuler tous ceux de date inférieure. Il calcule la prévision et envoie un message nul de prédiction uniquement durant la simulation du *arr non valide.* Dans l'état actuel de PARSEVAL, un simulateur envoie à tous ses successeurs la même prédiction, indiquant la date maximale avant laquelle il n'enverra pas de clients. La sélection sur les sorties (première étape de la méthode conservative) pour l'envoi des messages nuls est ainsi simplifiée, car le système n'est pas obligé de calculer une prévision pour chaque successeur. La date de la prédiction envoyée est celle du *arr non valide* augmentée de la prévision.

Le calcul de la prédiction dépend de quatre paramètres qui sont : la loi de service, le nombre de serveurs, la politique de gestion de la file et l'état courant du simulateur. Nous allons décrire ce calcul en tenant compte de ces paramètres.

`FCFS (First Come First Served):` premier arrivé - premier servi

Si *tous les serveurs sont indisponibles,* la date de la prédiction est celle du premier fs dans l'échéancier.

Dans *le cas contraire,* une durée de service est prétirée pour chacun des N serveurs libres, et les N premiers clients qui arriveront, se verront affectés ces services. La date virtuelle de fin de service, qui est la date du message nul reçu augmentée de la durée prétirée de service, est alors calculée pour chacun d'eux. La date de la prédiction est la date minimale entre ces dates virtuelles et celle du premier fs dans l'échéancier.

`LCFS (Last Come First Served)`: dernier arrivé - premier servi

Le simulateur est *préemptif* c'est à dire qu'un client qui vient d'arriver peut prendre le serveur occupé par un autre, et mettre celui-ci en attente. A partir de la date du *arr non valide*, aucun, un, deux ou une infinité de clients peuvent arriver, et leur durée de service respective est impossible à prétirer comme dans le cas du FCFS. Il est donc nécessaire de fixer une durée minimale de service qui peut être affectée à un nouvel arrivant. Dans PARSEVAL, la durée de service est minorée par cette limite au moment même de son tirage aléatoire. La date de la prédiction sera donc la plus petite entre celle du premier fs de l'échéancier, et celle du message nul reçu augmentée de la durée minimale de service.

`PRIORITÉ`: service selon les priorités

Lorsqu'un client de priorité p_{arr} arrive et qu'aucun serveur n'est disponible, il y a préemption si un des clients en service a une priorité strictement inférieure à p_{arr}. La démarche pour le calcul de la prévision est donc la suivante :

- Soit d_{fs} la date du premier événement **fs** de l'échéancier, et p_{ser} la priorité du client le moins prioritaire en service.
- On effectue un prétirage de service pour un future client de chacune des priorités supérieures à p_{ser}.
- La date de la prédiction est donc la plus petite entre d_{fs} et celles des événements **fs virtuels** correspondant aux prétirages de services effectués.

Dans PARSEVAL, le niveau de priorité d'un client peut être un entier quelconque. Il est donc impossible de suivre la démarche décrite précédemment car cela nécessite une quantité importante de mémoire. La prédiction est calculée de la même manière que dans le cas LCFS.

`QUANTUM`: les clients sont servis à tour de rôle. Chaque service élémentaire a une durée q fixée

Si *aucun serveur n'est disponible*, la solution la plus simple, la moins coûteuse en calcul, mais également la moins optimale, est de fixer la prédiction à la date du premier fs ou fq de l'échéancier. En fait, pour que la valeur de la prédiction soit maximale, il faut la calculer en faisant avancer fictivement la simulation, le plus loin possible dans le temps, et en tenant compte des clients en service, de ceux dans la file et de ceux qui peuvent arriver. Cette deuxième solution est plus efficace vis à vis des autres simulateurs pour le déblocage, mais pénalisante pour celui qui envoie car, plus la file d'attente est longue, plus le calcul est long. De plus, le cycle de rotation des clients dans le serveur peut s'étendre loin dans le temps, selon leurs durées résiduelles de service et la valeur du quantum (La durée résiduelle de service est le reste de temps que le client doit encore passer dans le serveur). Un compromis est donc à faire entre ces deux solutions, la deuxième étant la plus adaptée si chaque simulateur fonctionne sur un processeur qui lui est propre.

S'*il y a N serveurs libres*, une durée de service est prétirée pour chacun des N futurs nouveaux clients. Comme pour le cas précédent, la solution simple est de dire que la date de la prédiction est la date minimale entre celle du premier fs ou fq de l'échéancier, et celles des fins de service fictives pour ces nouveaux clients (date du message nul reçu augmentée de la durée de service prétirée). Pour la deuxième solution, les mêmes remarques s'appliquent également. Cette fois-ci, il faut, en plus, tenir compte des prétirages effectués. PARSEVAL utilise actuellement la première solution pour ces deux cas.

`PS (Processor sharing)`: partage du service par tous les clients présents

S'*il n'y a aucun client en service*, alors la date de la prédiction est celle du message nul reçu à laquelle s'ajoute la durée minimale de service. Cette durée minimale est la même pour tous les services tirés dans le système.

Cas où *il y a des clients en service* : la date de la prédiction est

- la plus petite date entre celle du premier événement *fs* et {*date courante* + *((durée minimale de service)* ∗ *(1* + *nombre de clients en service))*}, si la plus petite durée résiduelle de service, parmi celles de tous les clients, est supérieure à la durée minimale de service,
- la date du premier événement *fs* de l'échéancier.

2.2.8.6 Simulation des événements

La simulation d'un événement est la base des traitements dans le système (étape TRAITEMENT de la méthode conservative). Nous allons présenter chacune de ces simulations élémentaires sous forme d'algorithme, en fonction de la politique de gestion de la file d'attente, que nous abrègerons par "politique". L'événement en cours de simulation est toujours celui en tête de l'échéancier et dès qu'il a été traité, il est supprimé.

Si la politique est $LCFS$ ou *Priorité préemptive*, un client enlevé du serveur et remis dans la file d'attente pourra, lorsqu'il aura à nouveau le serveur, recommencer depuis le début (*restart*), ou reprendre le service à partir de son état au moment de la préemption (*resume*). Le choix entre ces deux possibilités est défini en même temps que la politique au début de la simulation.

Les étapes "tirage aléatoire de service" dans les algorithmes suivants tiennent compte des durées de service déjà prétirées antérieurement lors des traitements de messages nuls, ou déjà tirées et le service correspondant a été interrompu à la suite d'une préemption. Ces durées sont toujours minorées par la durée minimale de service.

`Arrivée`

La simulation d'un arr de date T consiste à insérer le nouveau client reçu dans la file, et, selon la politique, à générer un ds de même date dans l'échéancier. Si l'événement est un **arr non valide**, il est simplement supprimé de l'échéancier et n'est pas simulé.

```
Insérer le client dans la file d'attente
Si politique = LCFS ou politique = PS
alors
        Générer un ds de date T
sinon
        Si il y a un serveur disponible
        alors
                Générer un ds de date T
        sinon
                Si politique = PRIORITE
                alors
                        Si il y a un client moins prioritaire en service
                        alors
                                Générer un ds de date T
                        fin si
                fin si
        fin si
fin si
```

`Début de service`

Au début de la simulation d'un ds de date T:

- si la politique est $FCFS$ ou *Quantum*, il y a forcément un serveur libre,
- si la politique est $LCFS$ ou *Priorité*, il y une préemption si aucun serveur n'est libre,

- si la politique est PS, le problème de place dans le serveur ne se pose pas.

Cas

politique = $FCFS$
Si *file d'attente non vide*
alors
Prendre le premier client de la file
Tirer aléatoirement la durée de service du client
Mettre le client dans le serveur
Générer un fs de date (T + *durée de service*)
fin si
politique = $QUANTUM$
Si *file d'attente non vide*
alors
Prendre le premier client de la file
Tirer aléatoirement la durée de service du client
Mettre le client dans le serveur
Si *durée de service* <= q
alors
Générer un fs de date (T + *durée de service*)
sinon
Générer un fq de date ($T + q$)
fin si
fin si
politique = $LCFS$
Si *file d'attente non vide*
alors
Prendre le dernier client de la file
Si *aucun serveur disponible*
alors (préemption)
Recherche du plus ancien client en service
L'enlever du serveur et le remettre dans la file
Enlever le fs de ce client de l'échéancier
fin si
Tirer aléatoirement la durée de service du client
Mettre le client dans le serveur
Générer un fs de date (T + *durée de service*)
fin si
politique = $PRIORITE$
Si *file d'attente non vide*
alors
Prendre le client le plus prioritaire de la file
Si *aucun serveur disponible*
alors (préemption)
Recherche du client le moins prioritaire en service
L'enlever du serveur et le remettre dans la file
Enlever le fs de ce client de l'échéancier
fin si
Tirer aléatoirement la durée de service du client
Mettre le client dans le serveur
Générer un fs de date (T + durée de service)
fin si
politique = PS
Tirer aléatoirement la durée de service du client
Si *un seul client dans la file*
alors

Générer un fs de date (T + *durée de service*)
sinon
Enlever le seul fs de l'échéancier
Mettre à jour la durée résiduelle de service
de chaque client
Recherche de d : la durée résiduelle minimale de service
Générer un fs de date ($T + d$)
fin si
fin cas

`Fin de quantum`

Les opérations suivantes sont effectuées pour la simulation d'un fq de date T :

Enlever le client du serveur
Mettre à jour son résidu de service
Le remettre dans la liste
Générer un ds de date T

`Fin de service`

La sélection sur les sorties (étape 1 et 3 de la méthode conservative) pour l'envoi des clients est faite durant la simulation d'un fs de date T. Si la politique est PS, tous les clients sont en service, donc il n'est pas nécessaire de les ranger dans une autre liste que la file d'attente.

Tirer aléatoirement le destinataire
Si *politique* $\neq PS$
alors
Enlever le client du serveur
L'envoyer vers l'émetteur
Générer un ds de date T
sinon
Enlever le client de la file d'attente
L'envoyer vers l'émetteur
Mettre à jour le résidu de service de chaque client
Chercher le plus petit résidu de service
Générer un fs de date (*date courante* + *plus petit résidu de service*)
fin si

2.3 Validation par comparaison à QNAP2

QNAP2 [11] est un système de description, de manipulation et de résolution de modèles de réseaux de files d'attente. Il comprend un langage spécifique destiné à la description et au contrôle du modèle étudié, différents outils de résolution analytique et un module de simulation à événements discrets. C'est ce dernier module qui est utilisé pour la validation de notre simulateur distribué sur un réseau de transputers.

2.3.1 Les politiques de gestion de la file

Les politiques FCFS, LCFS, et Quantum sont identiques à celles implémentées dans QNAP2. Le seule différence est que dans QNAP2, les clients en service restent à la tête de la file tandis que dans

PARSEVAL, ils sont mis dans une liste qui représente les serveurs. Il suffit tout simplement de tenir compte de cette différence dans la représentation et l'interprétation des résultats de simulation.

`FCFS (First Come First Served)`

Le premier client arrivé dans la file est le premier servi. Un client qui occupe un serveur le garde jusqu'à la fin de son service.

`LCFS (Last Come First Served)`

Le dernier client arrivé dans la file est servi immédiatement. Dans le cas où aucun serveur n'est disponible, il y a préemption. Le client privé de serveur reprendra son service au point de rupture (*resume*) ou depuis le début (*restart*), seulement après la fin des services de ceux arrivés après lui.

`QUANTUM`

Chaque client présent dans la file est servi à tour de rôle par tranche de temps fixe (*quantum*), l'ordre de services étant de type premier arrivé premier servi.

`PS (Processor Sharing)`

Tous les clients arrivés se partagent le serveur et sont donc servis en même temps.

`PRIORITÉ`

Les clients sont servis par ordre de priorité. Il y a préemption si un client plus prioritaire arrive après un autre moins prioritaire déjà en service. La reprise de service peut être du type *resume* ou du type *restart* comme dans le cas du LCFS. Actuellement dans PARSEVAL, c'est le générateur qui fixe la priorité d'un client dès sa création, et cette priorité ne change pas jusqu'à son élimination (dans QNAP2, les priorités peuvent être gérées dynamiquement).

2.3.2 Les lois de service

La durée de service d'un client est une variable aléatoire X qui suit une des lois ci-après. Sa valeur est calculée à partir de tirages uniformes U_i entre 0 et 1.

`Loi constante` : $X = CST(X_0)$

X a une valeur constante X_0 indentique pour tous les clients.

$$X = X_0.$$

`Loi uniforme` : $X = UNIF(a, b)$

X a une valeur uniforme entre les deux paramètres a et b.

$$X = a + U(b - a).$$

`Loi exponentielle` : $X = EXP(m)$

Cette loi a une fonction de distribution $F(x) = 1 - e^{-\lambda x}$ où λ est le paramètre et $m = 1/\lambda$ la moyenne. Par inversion, on obtient $X = -\ln(1 - U)/\lambda$.

$(U - 1)$ et U ayant la même distribution,

$$X = -m \ln(U).$$

`Loi hyper-exponentielle` : $X = HEXP(m, a)$

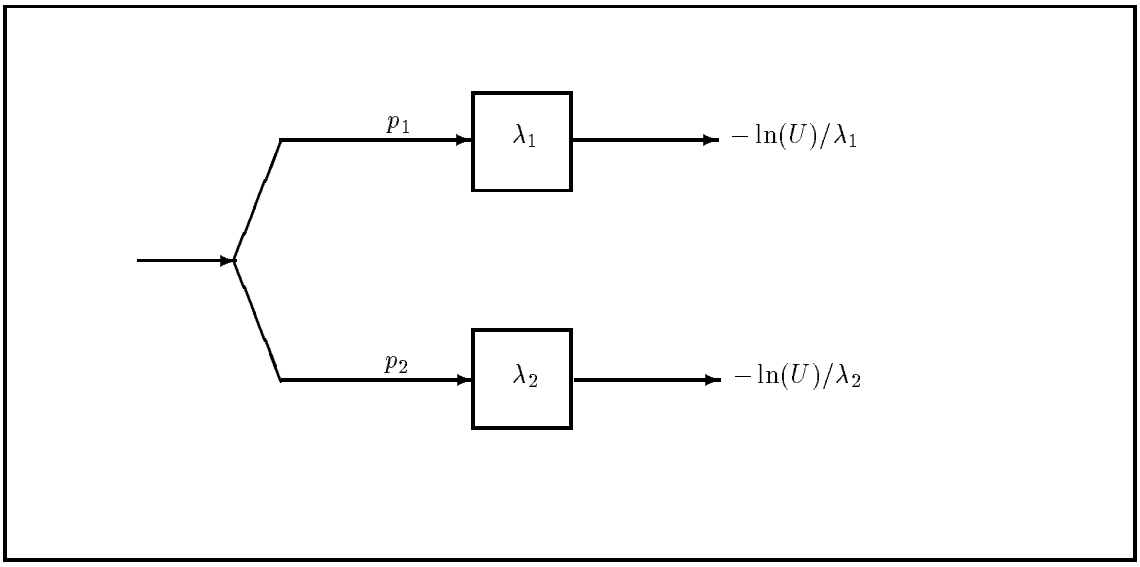

Figure 2.9: La loi hyper-exponentielle

Une variable aléatoire X de loi hyper-exponentielle est choisie parmi k variables aléatoires indépendantes de loi exponentielle, à chacun d'eux étant associée une probabilité p_i de choix, i variant de 1 à k. m est la moyenne et a le carré du coefficient de variation. Ici, nous nous limitons à une loi hyper-exponentielle à deux étages qui permet d'avoir un large coefficient de variation (figure 2.9), et à des valeurs entières de a strictement supérieures à 1. La procédure de calcul de X est la suivante :

Si $a \leq 1$
  alors sortie erreur
  sinon
    Tirage d'un U uniforme entre 0 et 1
    Si $U \leq \frac{1}{2a-1}$
      alors X suit une loi exponentielle de moyenne $m * a$
      sinon X suit une loi exponentielle de moyenne $m/2$
    fin si
fin si

`Loi d'Erlang` : $X = ERLANG(m, k)$

Une variable aléatoire qui suit une loi d'Erlang de moyenne m et de k *étages* est la somme de k variables aléatoires indépendantes de même distribution exponentielle, et chacune de paramètre λ. La valeur de X déduite de la loi exponentielle décrite plus haut est donc $X = -\sum_{i=1}^{k} \ln(U_i)/\lambda$. Cette expression peut être simplifiée et donne

$$X = -(m/k)\ln(\prod_{i=1}^{k} U_i), \quad m = k/\lambda.$$

Il faut donc effectuer i tirages uniformes indépendants pour le calcul de X.

`Loi de Cox` : $X = COX(a_1, b_1, a_2, b_2, ..., a_{n-1}, b_{n-1}, a_n, 0)$

Une variable aléatoire suivant une loi de Cox est également la somme de k variables aléatoires indépendantes chacune de loi exponentielle. Contrairement à la loi d'Erlang, la valeur de k n'est pas constante ($1 \leq k \leq n$) et les k variables aléatoires ne sont pas forcément identiques. a_i est la moyenne

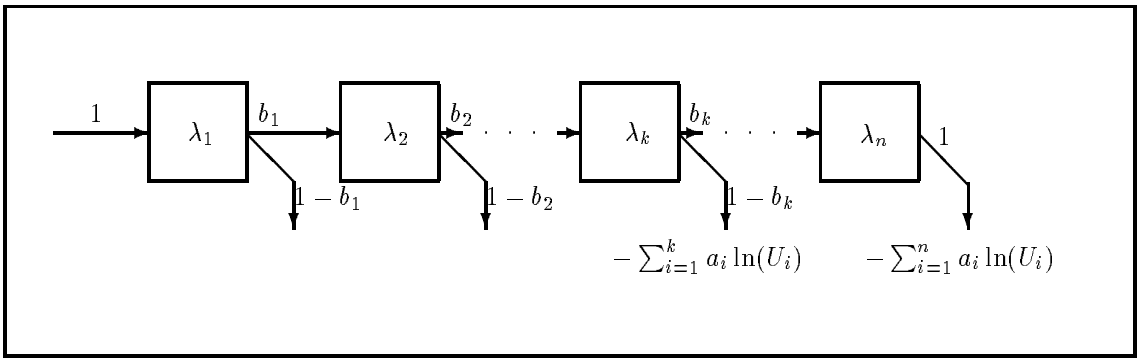

Figure 2.10: La loi de Cox

de la loi exponentielle du i-ième étage et b_i la probabilité de passer du i-ième étage au $(i+1)$-ième étage.

$$X = -\sum_{i=1}^{k} a_i \ln U_i, \qquad 1 \leq k \leq n.$$

Il faut donc effectuer le tirage des U_i et le tirage du nombre d'étages. Comme les lois de passage entre les étages sont indépendantes, il est possible de faire un cumul des probabilités pour chaque étage, de normaliser ces cumuls et d'effectuer un seul tirage uniforme pour obtenir le nombre d'étages.

2.3.3 Les résultats

Lorsque la simulation se termine normalement c'est à dire que la date finale a été atteinte par tous les simulateurs, chacun d'entre eux envoie à l'écran pour affichage les résultats statistiques suivants sur l'ensemble de la simulation (entre parenthèses est indiqué le nom du résultat respectivement dans PARSEVAL et dans QNAP2) :

- le nombre de messages nuls envoyés (NUL, *sans objet*),
- le nombre total de clients arrivés dans la file (ARR, non représenté),
- le nombre total de clients servis et envoyés (SERV, SERV NB),
- la durée moyenne de service (MOY.SERV, SERVICE) qui est la moyenne de tous les services tirés aléatoirement,
- la longueur moyenne de la file par unité de temps (ACC, CUST NB), calculée à partir de la variation dans le temps (selon les dates virtuelles de simulation) du nombre de clients dans la file, et
- le temps moyen d'occupation d'un serveur par unité de temps (TPS.OCC, BUSY PCT), qui est la moyenne des temps d'occupation de tous les serveurs.

Le nombre de messages nuls donne des indications sur la synchronisation des stations, et les autres résultats donnent les statistiques sur la simulation. Les modules statistiques ([13], [14]) n'ayant pas encore été implémentés, PARSEVAL ne fournit pas des informations sur la précision de ces mesures.

L'exemple suivant est un modèle qui regroupe différentes politiques de gestion de files et différentes lois de service dans un même modèle. C'est un réseau simple composé de six stations en anneau (figure 2.11), simulé entre les dates virtuelles de simulation 0 et 100000. Le nom, la politique de gestion et la loi de service de chaque station sont montrés par la figure et la source génère 150 clients. A la fin de la simulation, tous les clients sont sortis du réseau et ont tous été servis par chacune des stations.

Les résultats de PARSEVAL :

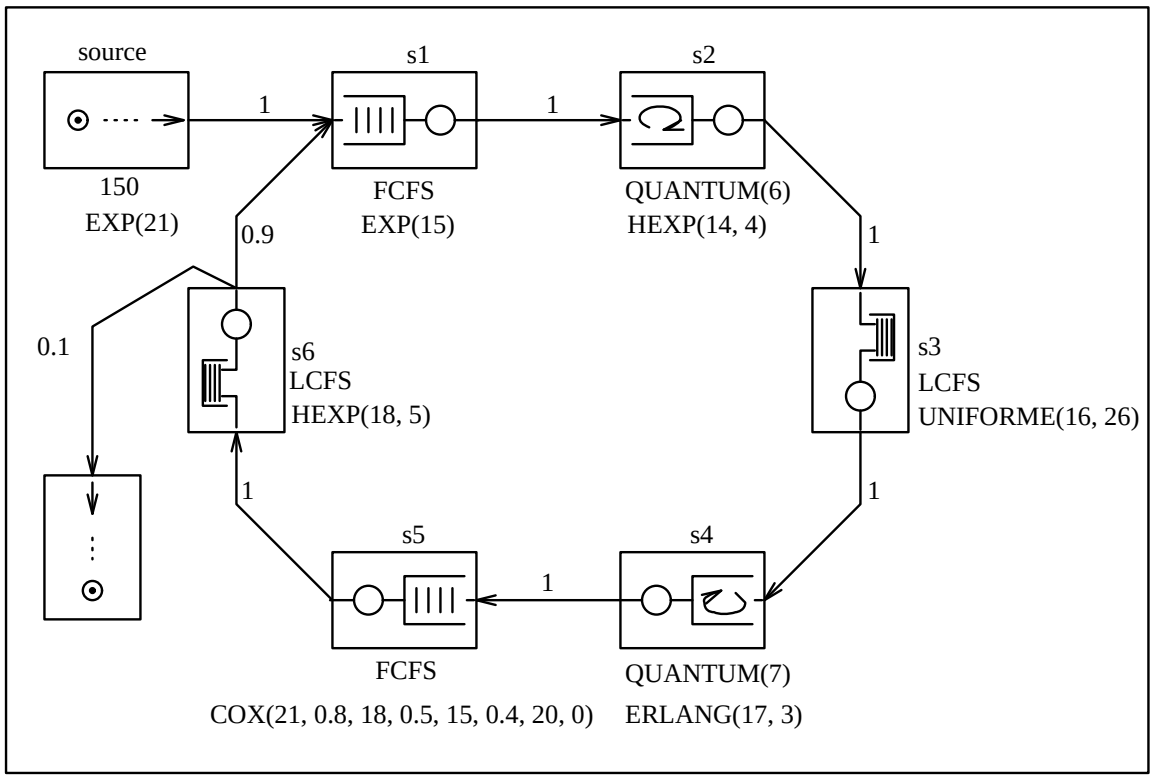

Figure 2.11: Exemple de comparaison avec QNAP2

NAME	*ARR*	*SERV*	*NUL*	*MOY.SERV*	*ACC*	*TPS.OCC*
source		150				
s1	1388	1388	3605	14.61763	0.51336	0.20256
s2	1388	1388	3750	12.83339	0.61855	0.17818
s3	1388	1388	4577	21.02911	2.51644	0.29186
s4	1388	1388	5242	16.61463	0.77006	0.23071
s5	1388	1388	2783	43.64620	37.64723	0.60584
s6	1388	1388	2652	16.52014	0.31854	0.22930

Les résultats de QNAP2 :

NAME	*SERVICE*	*BUSY PCT*	*CUST NB*	*RESPONSE*	*SERV NB*
SRC	21.2900	0.3193E-01	2.2390	1493.000	150
+/-	3.7860	0.1125	5.2120	455.200	
S1	14.9800	0.2128	0.5050	35.540	1421
+/-	0.9096	0.1146	0.6581	32.910	
S2	14.6600	0.2083	0.3828	26.940	1421
+/-	1.7020	0.1206	0.2889	11.070	
S3	21.0700	0.2994	2.7820	195.800	1421
+/-	0.1601	0.1476	16.3200	565.300	
S4	17.1300	0.2435	0.4049	28.500	1421
+/-	0.4806	0.1263	0.3392	7.733	
S5	44.9400	0.6386	35.9400	2529.000	1421
+/-	2.4880	0.2314	25.3300	724.000	
S6	17.4200	0.2476	0.3941	27.740	1421
+/-	2.6090	0.5269E-01	0.1754	7.928	

Ces deux tableaux montrent que les résultats fournis par PARSEVAL convergent vers les résultats de QNAP2 pour un niveau de confiance de 95% (voir [11]). Plusieurs tests de comparaison de même type que cet exemple (avec des réseaux plus complexes contenant plus de boucles) ont été effectués pour la validation des politiques de gestion et des lois de service de PARSEVAL et ont montré la correction des résultats.

Chapitre 3

Les performances

Les points suivants sont la base des performances du système :

- Dans la version actuelle de PARSEVAL, un simulateur d'une file d'attente est une unité de compilation qui ne peut être distribuée sur plusieurs processeurs. Le placement optimal du point de vue charge de calcul (simulation) est donc un seul simulateur sur un processeur si le nombre de processeurs est suffisant.
- La gestion des communications sur les liens physiques du transputer est assurée intégralement par le gestionnaire de lien. L'unité centrale est donc totalement déchargée de cette tâche [1].
- Le temps perdu durant les blocages des serveurs dépend de la répartition des messages circulant, dans le temps réel.
- Le temps d'acheminement d'un message dépend très fortement de la localisation de l'emetteur et du récepteur.

3.1 La charge de calcul

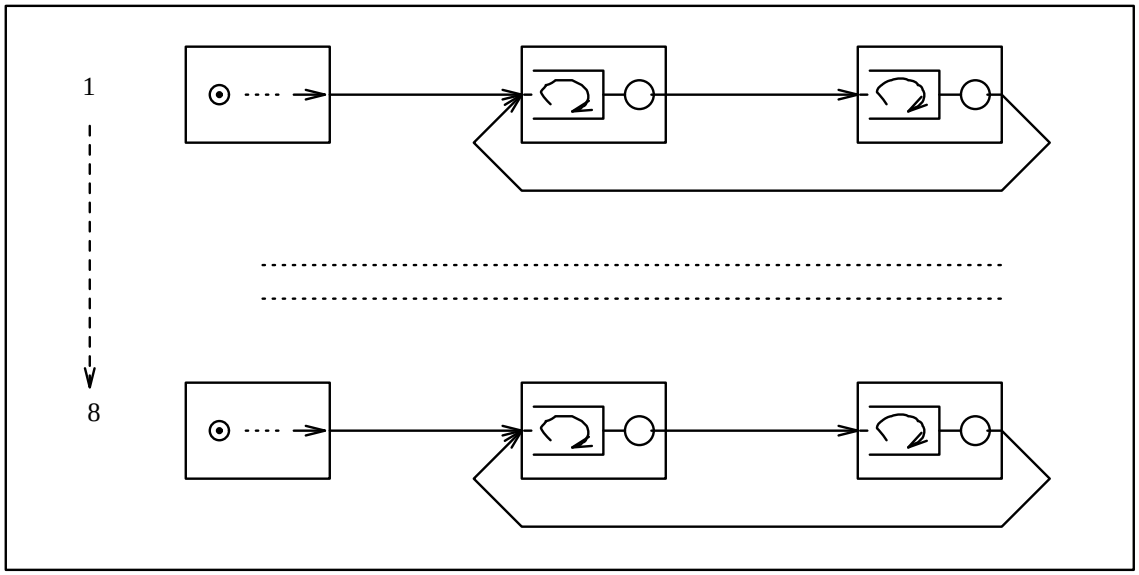

Figure 3.1: Huit modèles indépendants

La figure 3.1 montre huit modèles identiques de réseau, chacun composé de trois processus : une source de clients et deux stations, chacune de loi de service HEXP(15, 2) et de politique de gestion

[1] à l'exception d'un accès mémoire tous les 4 octets envoyés

QUANTUM(1). L'ensemble des trois processus d'un réseau est placé sur un même processeur, ainsi il n'y a pas de communication entre les processeurs et la simulation de chacun des modèles représente la même charge de calcul, le coût des communications internes étant pris en compte comme un coût de calcul. Chacune des stations sert environ 6500 clients et génère 99000 messages nuls. La durée de la simulation en fonction du nombre p de processeurs ($1 \leq p \leq 8$) est montré par la figure 3.2. Différents tests du même type ont été réalisés et ont donné le même résultat.

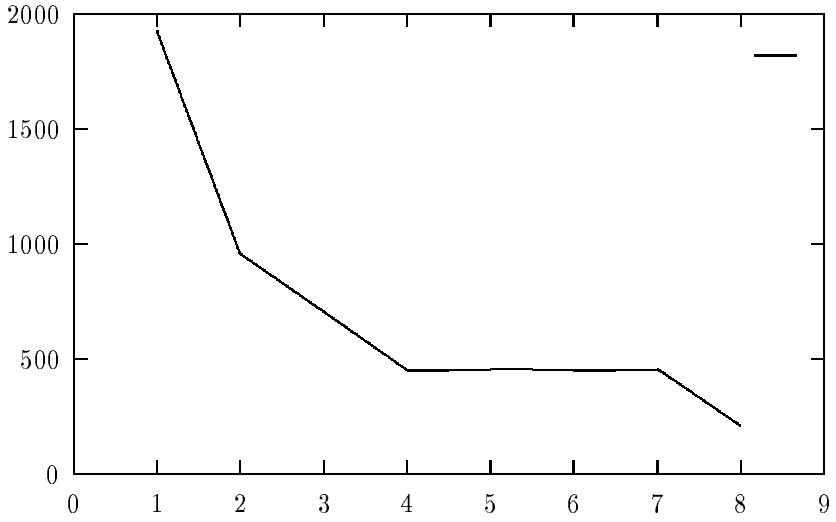

Figure 3.2: Durée de simulation en seconde du modèle de la figure 3.1

Si on dispose donc de p processeurs pour simuler n modèles identiques, chacun représentant une charge unitaire de calcul u, et ne communiquant pas avec les autres, la durée globale de la simulation correspond approximativement à la durée de l'exécution sur le processeur le plus chargé. Cette charge est

$$c = \lceil n/p \rceil u \quad \text{pour } 1 \leq p \leq n.$$

3.2 Le coût des communications

Les résultats ci-dessus restent toujours valables du point de vue charge de calcul. Les communications, bien qu'un peu moins coûteuses par rapport aux calculs si elles ne sont pas trop importantes (les communications internes occupent peu de temps CPU et les communications externes sont assurées par le gestionnaire de liens), introduisent des périodes d'inactivité (idle) des processeurs et des périodes de blocage des simulateurs à cause de la vitesse de transmission sur les liens physiques.

On place les stations du modèle précédent de telle manière que certains liens du modèle soient affectés à des liens physiques du réseau de transputers, créant ainsi des communications entre les processeurs. Par exemple, la deuxième station d'un sous-réseau A est placée sur le processeur où se trouve le sous-réseau B, et la deuxiéme station de ce dernier sur le même processeur que A. Les communications entre les deux stations de A et entre les deux stations de B passent donc entièrement par le réseau de routage en anneau de PARSEVAL. Huit transputers sont utilisés et chacun d'eux simule une source et deux stations.

La figure 3.3 montre des courbes de vitesses de simulation (en secondes) en fonction du nombre de clients servis par une station. 2, 4, 6 et 8 sont les nombres de sous-réseaux qui font circuler leurs messages sur l'anneau de routage, 0 étant le cas où aucun message ne passe par cet anneau. Sachant que, dans cet exemple, le nombre de messages nuls générés par chacune des stations est 14 *fois* le nombre x de clients servis, un sous-réseau fait donc passer $15x$ messages sur l'anneau de routage, multipliant ainsi la vitesse de simulation par 3.5 environ.

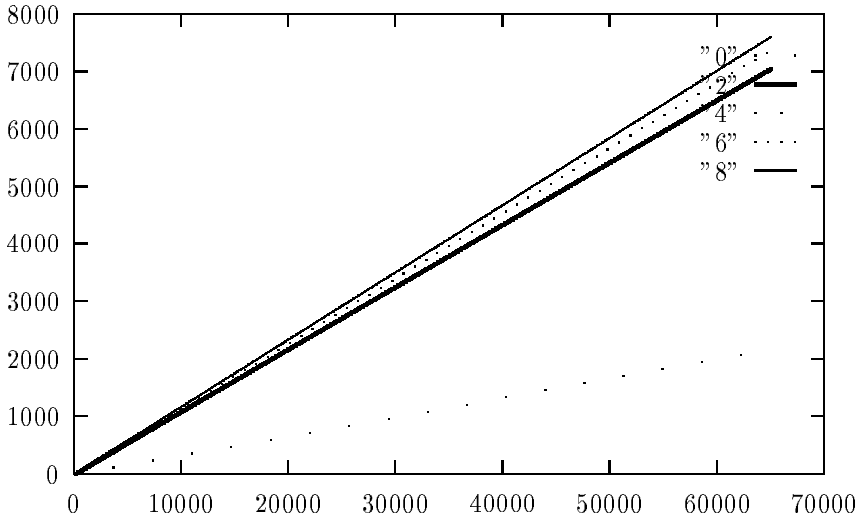

Figure 3.3: Durées de simulation en fonction de la quantité de communications

Le peu de différence entre les courbes 2, 4, 6 et 8 s'explique par le fait que :

- d'une part, l'anneau de routage n'est pas encore saturé, et
- d'autre part, la durée totale de simulation qui est la quantité mesurée ici, est la durée la plus longue entre celles de simulation de chacune des stations.

Dans le cas général, la durée des tranches de temps perdus en attente de communication dépend :

- de la configuration du réseau physique de transputers,
- du nombre de files, de liens et de boucles dans le modèle,
- des paramètres de chacune des stations : la politique de gestion, la loi de service et le nombre de serveurs,
- du placement des processus sur les processeurs, et
- de la vitesse réelle de simulation de chacune des files.

On ne peut agir que sur le placement des processus sur les processeurs pour minimiser ces temps d'inactivité. Ce placement génère une certaine quantité de messages nuls qui, selon leur nombre, minimiseront les temps de blocage ou compromettront les performances de chacun des processeurs et de l'ensemble, la transmission sur les liens physiques du transputer étant relativement lente selon la quantité de messages à envoyer bien que l'unité centrale en soit totalement déchargée. Dans le cas où les processus communiquent au sein d'un même transputer, un nombre important ou peu élevé de messages nuls n'est pas forcément une pénalisation. Dans le cas contraire, une forte synchronisation des processus peut rapidement devenir très pénalisante si la charge en calcul de chacun des processeurs en faible.

L'amélioration des performances se résume donc au problème suivant : trouver, pour un modèle et un réseau donné de transputers, un placement optimal des processus, qui minimise la durée de simulation en

- partageant correctement la charge de calcul, et en
- générant la quantité optimale de messages nuls,

sachant que la parallélisation améliore les performances pour certains modèles, et ne les change pas sinon les diminue pour d'autres. Ceci est dû à la structure même et au degré de parallélisme intrinsèque du modèle [7].

3.3 Comparaison avec QNAP2

La simulation par QNAP2 de l'ensemble des huit réseaux de la figure 3.1 dure 1600 *secondes* sur une station de travail de type SUN3-60. Le même modèle est simulé en 1930 *secondes* sur un transputer et en 960 *secondes* sur deux transputers non communiquant (figure 3.2). On peut donc espérer que dans le cas de la simulation par réplication, dans la limite de la mémoire de chaque transputer, PARSEVAL simule plus vite que QNAP2 dès qu'on utilise plus d'un processeur.

Nous allons considérer d'autres exemples pour bien estimer les performances selon le type du modèle à simuler. La figure 3.4 montre quatre types d'ateliers de production : *atelier*1 est un ensemble de stations de travail disposées en arbre, *atelier*2 en réseau complexe, *atelier*3 en linéaire et *atelier*4 en anneau.

Les figures 3.5, 3.6, 3.7 et 3.8 montrent la durée de simulation de ces quatre réseaux par PARSEVAL en fonction du nombre de processeurs, le placement des processus étant optimal pour chacun de ces nombres. Chaque station de *atelier*1 sert environ 80 clients, celle de *atelier*2 3500 clients, celle de *atelier*3 110 clients et celle de *atelier*4 1160 clients. QNAP2 simule *atelier*1 en 3.6 *secondes* environ, *atelier*2 en 197 *secondes*, *atelier*3 en 7 *secondes* et *atelier*4 en 16.6 *secondes*.

On constate que pour des modèles sans boucle donc générant moins de messages nuls, PARSEVAL simule plus vite que QNAP2 même avec un seul transputer. Dès que le modèle contient des boucles, comme dans le cas de *atelier*2 et *atelier*4, il lui faut quelques processeurs pour battre QNAP2, en l'occurrence 6 transputers. Pour *atelier*2, il est inutile d'utiliser plus de 6 *processeurs* car la structure même de l'atelier, associée au système de routage en anneau de PARSEVAL ne permet pas d'accélérer la simulation. Le placement optimal est chaque partie P_i (figure 3.4) sur un processeur. Avec un autre système de routage, il est possible d'améliorer les performances en utilisant plus de 6 processeurs, par exemple la partie P_6 pourrait être partagée sur deux processeurs car elle est composée de deux sous-réseaux indépendants.

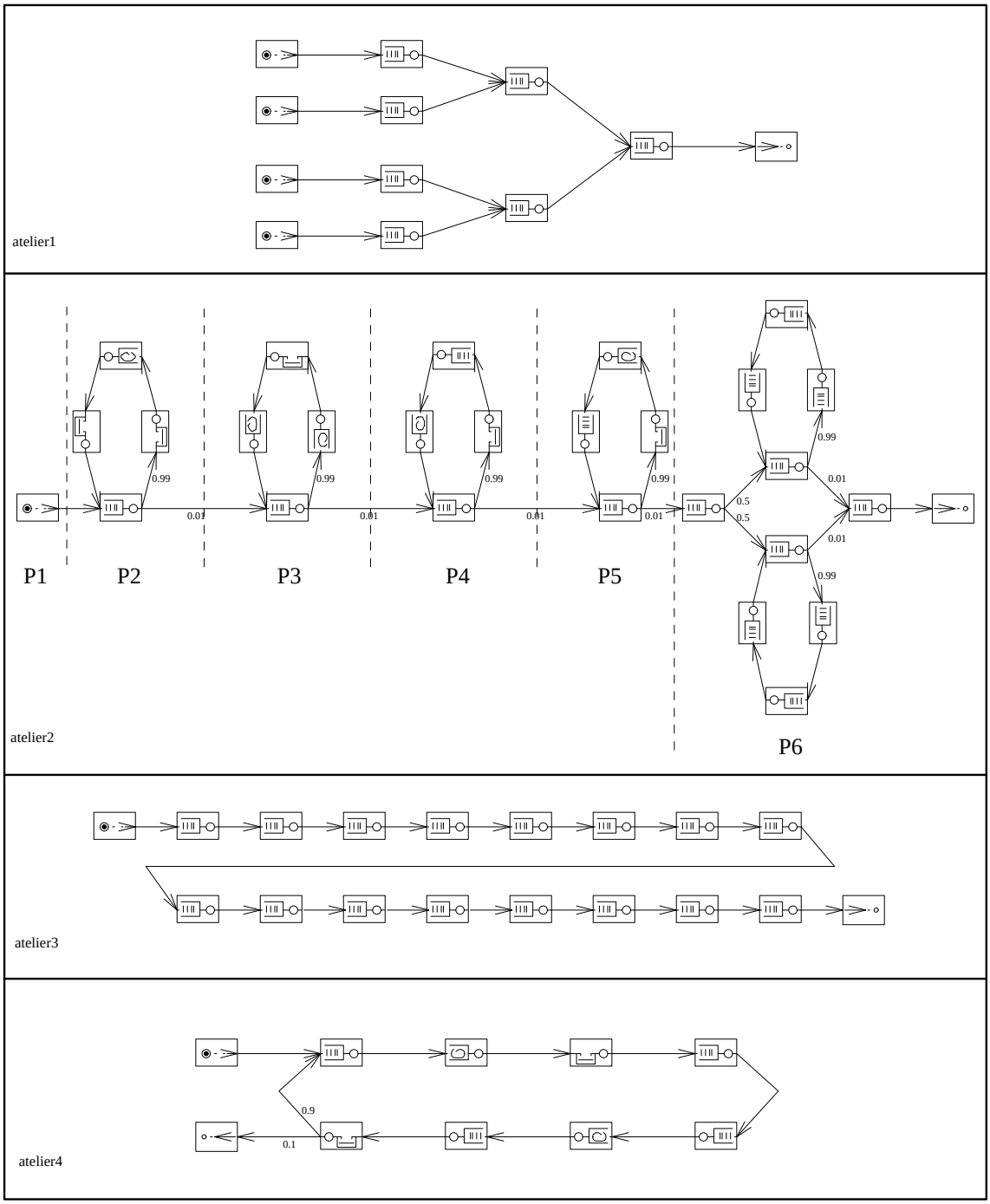

Figure 3.4: Exemples d'atelier de production

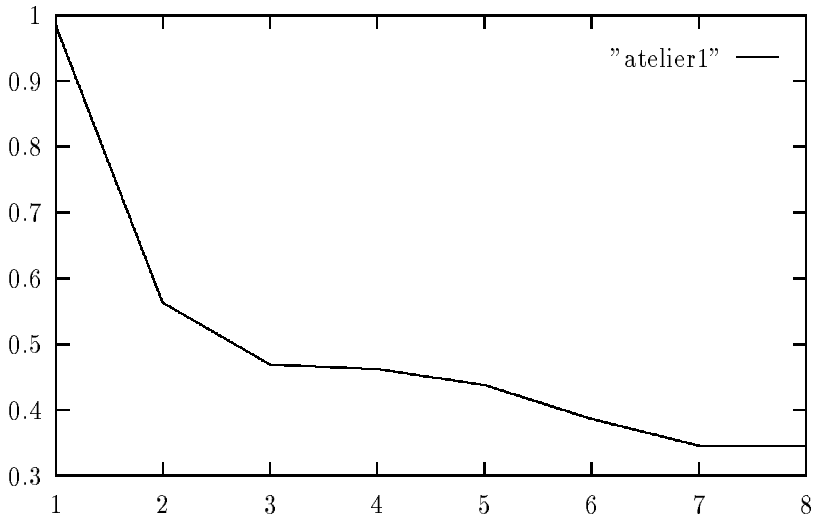

Figure 3.5: Duree de simulation de atelier1

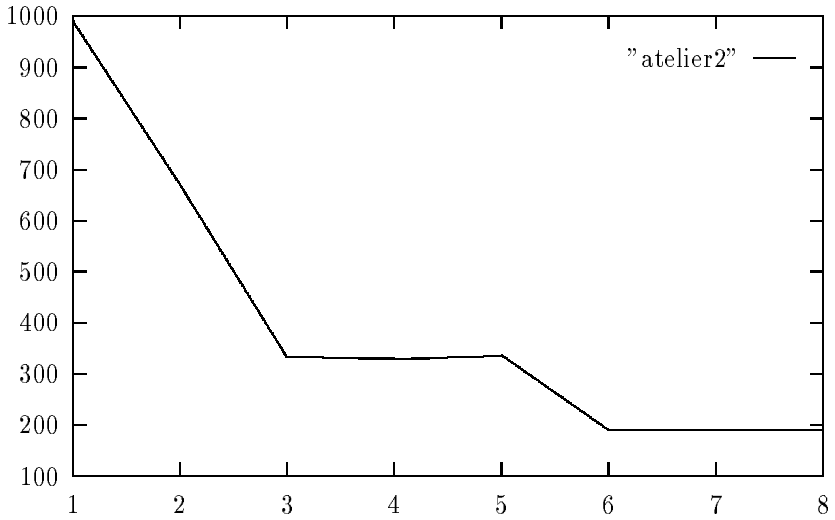

Figure 3.6: Duree de simulation de atelier2

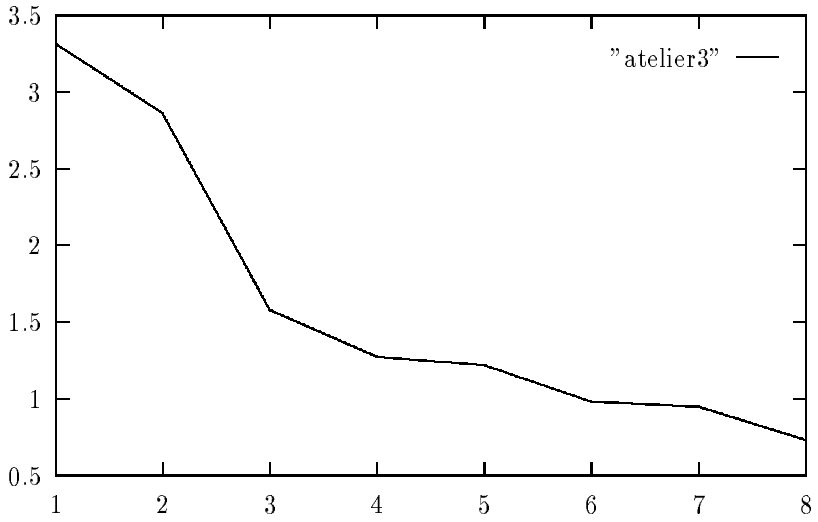

Figure 3.7: Duree de simulation de atelier3

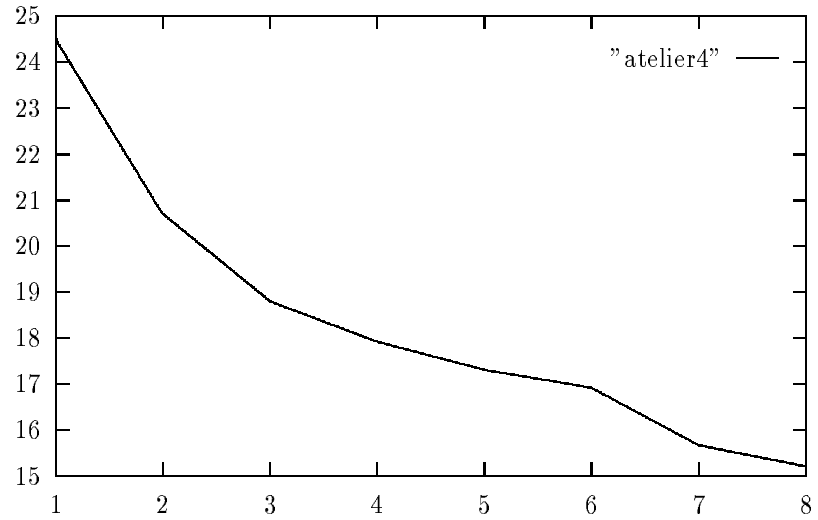

Figure 3.8: Duree de simulation de atelier4

Conclusions

On constate que la vitesse d'exécution de PARSEVAL n'est pas satisfaisante pour les modèles à boucle bien qu'elle soit correcte pour les autres modèles. Il est donc nécessaire d'apporter les améliorations suivantes pour utiliser au maximum les processeurs.

- Utiliser un réseau de routage plus adapté que le réseau en anneau actuel, et optimiser le routeur.
- Optimiser le code des serveurs et éventuellement le fragmenter en plusieurs parties pouvant être distribuées sur plusieurs processeurs. Ceci permettra une répartition plus fine des processus au début de la simulation et simplifiera le problème de leur placement dynamique car la granularité du code à transférer est augmentée.
- Optimiser la politique d'envoi de messages nuls. Différentes possibilités sont envisageables :
  - envoi d'un message nul seulement après un "timeout" de blocage,
  - seuls les serveurs qui ont plus de deux entrées en génèrent, les autres ne font que les faire passer.
- Fixer la durée minimale de service selon les paramètres de la loi pour augmenter la prévision.
- Occuper le serveur pendant les périodes de blocage, par exemple en faisant le maximum de prétirages de service.
- Ajouter un module de pré-étude du modèle pour définir jusqu'où il est parallélisable.

La méthode optimiste dite "*time warp*" doit être implémentée sous les mêmes conditions et comparée à la méthode conservative étudiée ici.

Bibliographie

[1] J. LEROUDIER. *La simulation à événements discrets.*
Editions Hommes et Techniques - 1980.

[2] K. M. CHANDY - J. MISRA. *Asynchronous Distributed Simulation via a Sequence of Parallel Computation.*
Communications of the ACM - Vol.24, n.11, Avril 1981.

[3] K. M. CHANDY - J. MISRA. *Distributed Simulation : A Case Study in Design and Verification of Distributed Programs.*
IEEE Transactions on Software Engineering - Vol.SE-5, n.5, Sept 1979.

[4] J. MISRA. *Distributed Discrete-event Simulation.*
Computing Surveys - Vol.18, n.1, Mars 1986.

[5] D. R. JEFFERSON. *Virtual Time.*
ACM Transactions on Programming Langages and Systems. Vol.7, n.3,Juill. 1985.

[6] D. R. JEFFERSON. *Fast Concurrent Simulation Using the Time Warp Mechanism.*
Rand Corporation, Santa Monica, California.

[7] D. B. WAGNER - E. D. LAZOWSKA. *Parallel Simulation of Queueing Networks: Limitations and Potentials.*
Performance Evaluation Review, Vol. 17 #1, Mai 1989.

[8] W.-K. SU - C. L. SEITZ. *Variants of the Chandy-Misra-Bryant Distributed Discrete-event Simulation Algorithm.*
Distributed Simulation 1989, Simulation Series, Vol. 21, No. 2.

[9] W. CAI - S. J. TURNER. *An Algorithm for Distributed Discrete-event Simulation - The "Carrier Null Message" Approach.*
Distributed Simulation, Jan. 1990, Simulation Series, Vol. 22, No. 1.

[10] D. E. KNUTH. *The Art of Computer Programming.*
Vol.2, Second Edition.

[11] D. POTIER - M.VERAN. *QNAP2 : A Portable Environment for Queueing Systems Modelling.*
Rapport de recherche n.314, INRIA Rocquencourt, Juin 1984.

[12] SIMULOG. *QNAP 2 Reference Manual.*

[13] P. BRATLEY - B. L. FOX - L. E. SCHRAGE. *A Guide to Simulation.*
Second edition, Springer-Verlag 1983.

[14] K. PAWLIKOWSKI. *Steady-State Simulation of Queueing Processes: A Survey of Problems and Solutions.*
ACM Computing Surveys, Vol. 22, No. 2, Juin 1990.

[15] INMOS 1988. *Transputer Data Book.*

[16] INMOS 1988. *OCCAM 2 Reference Manual.*

Table des matières